

*NASA Conference Publication 10052*

# **NASA Formal Methods Workshop 1990**

(NASA-CP-10052) NASA FORMAL METHODS  
WORKSHOP, 1990 (NASA) 504 p CSCL 098

N91-17559

--THRU--

N91-17578

Unclas

G3/61 0319681

*Compiled by  
Ricky W. Butler  
NASA Langley Research Center  
Hampton, Virginia*

Proceedings of a workshop sponsored by the  
National Aeronautics and Space Administration,  
Washington, D.C., and held at  
Langley Research Center  
Hampton, Virginia  
August 20-23, 1990

**November 1990**

**NASA**

National Aeronautics and  
Space Administration

**Langley Research Center**  
Hampton, Virginia 23665-5225

## Contents

Introduction .....	1
Workshop Agenda .....	2
List of Attendees .....	5
Digital Avionics a Cornerstone of Avionics by Chuck Meissner for Cary Spitzer (NASA) .....	6 <sub>51</sub>
Life Critical Digital Flight Control Systems by James McWha (Boeing Military) .....	30 <sub>52</sub>
Advanced Embedded Processing: Present and Future by Jerry Cohen (Boeing Military) .....	52 <sub>53</sub>
MAFT: The Multicomputer Architecture for Fault Tolerance by Roger Kieckhafer (U. Nebraska, Lincoln) .....	94 <sub>54</sub>
Design For Validation by Rick Butler (NASA) .....	146 <sub>55</sub>
What FM can offer to DFCS design by John Rushby (SRI International) .....	165 <sub>56</sub>
What FM can offer to DFCS design by Don Good (Computational Logic, Inc) .....	195 <sub>57</sub>
High Level Design Proof of a Reliable Computing Platform Ben DiVito (Vigyan) .....	232 <sub>58</sub>
A Verified Model of Fault Tolerance John Rushby (SRI International) .....	260 <sub>59</sub>
The Design and Verification of a Fault-tolerant Circuit Bill Bevier and Bill Young (Computational Logic, Inc) .....	280 <sub>510</sub>
Verifying an Interactive Consistency Circuit	



by Mandayam Srivas (Odyssey Research Associates) .....	295	511
Hardware Verification at Computational Logic Inc. by Warren Hunt (Computational Logic, Inc) .....	325	512
Generic Interpreters and Microprocessor Verification by Phil Windley (Univ. California, Davis) .....	347	513
VIPER Project by Clive Pygott and John Kershaw (Royal Signals and Radar Establishment) .....	380	514
Mechanical Proofs of Fault-Tolerant Clock Synchronization by N. Shankar and John Rushby (SRI International) .....	405	515
An HOL Theory For Voting by Paul Miner and Jim Caldwell (NASA) .....	442	516
Formally Specifying the Logic Of Automatic Guidance Control (Ada) by David Guaspari (Odyssey Research Associates) .....	457	517
Verification of Floating-point Software by Doug Hoover (Odyssey Research Associates) .....	477	518
C Formal Verification with Unix Communication and Concurrency by Doug Hoover (Odyssey Research Associates) .....	495	519

## Introduction

This publication contains copies of the material presented at the NASA Formal Methods Workshop held at Langley Research Center on August 20-23, 1990. The purpose of the workshop was to bring together the researchers involved in the NASA formal methods research effort for detailed technical interchange and to provide a chance for interaction with representatives from the U.S. government and the aerospace industry. The goals of the workshop were:

- Introduce the formal methods research teams to a broader view of the aerospace problem domain by industry presentations.
- Detailed technical exchange between formal methods research teams to define and characterize the verification problem for ultra-reliable life-critical flight control systems.
- Identification of aerospace problems which can benefit from formal methods and can serve as the basis of future research efforts.

The NASA effort in formal methods includes researchers at NASA LaRC, Computational Logic Inc., Odyssey Research Associates, SRI International, Boeing Military, Vigyan and the University of California at Davis and Irvine. Also NASA Langley is involved in a joint research effort with the UK Royal Signals and Radar Establishment as formalized in a Memorandum of Understanding between the two organizations.

Attendees at the workshop included NASA personnel, researchers from the four supporting contract organizations, RSRE personnel, invited speakers, and representatives from other government research organizations with interests in formal methods. Attendance was by invitation only.

**NASA Formal Methods Workshop Agenda**  
**(Aug 20-23, 1990)**

**Day 1**

8:00 - 8:20 am		Late Registration
8:20 - 8:30 am	Milt Holt	Greeting by Chief of ISD
8:30 - 8:45 am	Ricky W. Butler	Workshop Objectives
8:45 - 9:30 am	Chuck Meissner	Digital Avionics: A Cornerstone of Aviation
9:30 - 10:15 am	James McWha	Life Critical Digital Flight Control Systems (DFCS)
10:15 - 10:30 am	BREAK	
10:30 - 11:30 am	Jerry Cohen	Advanced Embedded Processing: Present and Future
11:30 - 12:30 am	LUNCH	
12:30 - 1:30 pm	Roger Kieckhafer	MAFT: The Multicomputer Architecture for Fault Tolerance
1:30 - 2:00 pm	Rick Butler	Design For Validation
2:00 - 2:30 pm	BREAK	
2:30 - 3:00 pm	Richard Platek	What FM can offer to DFCS design
3:00 - 3:30 pm	John Rushby	What FM can offer to DFCS design
3:30 - 4:00 pm	Don Good	What FM can offer to DFCS design
7:00 pm	Conference Dinner	

## Day 2

### — OS verification —

8:30 - 9:30 am	DiVito	High Level Design Proof of a Reliable Computing Platform
9:30 - 10:15 am	Rushby	A Verified Model of Fault Tolerance
10:15 - 10:30 am	BREAK	

### — Byzantine Generals —

10:30 - 11:30 pm	Bevier & Young	The Design and Verification of a Fault-tolerant Circuit
11:30 - 12:30 am	LUNCH	
12:30 - 1:30 am	Srivas	Verifying an Interactive Consistency Circuit

### — Microprocessor —

1:30 - 2:00 pm	Hunt	Hardware Verification at Computational Logic Inc.
2:00 - 2:30 pm	Windley	Generic Interpreters and Microprocessor Verification
2:30 - 3:00 pm	BREAK	
3:00 - 4:00 pm	Pygott/Kershaw	VIPER 2 & NODEN
4:00 - 4:30 pm	Discussion	

### Day 3

— Clock Synchronization —		
8:30 - 10:00 am	Shankar/Rushby	Mechanical Proofs of Fault-Tolerant Clock Synchronization
10:00 - 10:30 am	Discussion	
— Commercial Chips —		
10:30 - 11:30 pm	Levitt	Floating-pt. Coprocessor (Intel 8087) DMA controller (Intel 8237A), etc. (Not in Proceedings)
11:30 - 12:00 pm	Caldwell/Carreno/ Miner	An HOL Theory For Voting
12:00 - 1:00 pm	LUNCH	
— Code Verification —		
1:00 - 1:45 pm	Guaspari	Formally Specifying the Logic Of Automatic Guidance Control (Ada)
1:45 - 2:30 pm	Hoover	Verification of Floating-point Software
2:30 - 3:00 pm	Hoover	C Formal Verification with Unix Communication and Concurrency
3:00 - 3:30 pm	BREAK	
3:30 - 5:00 pm	Planning	

### Day 4

8:30 - 12:00 pm Discussion

## NASA FM Workshop Attendees

Bill Bevier	CLI	(512)322-9951	bevier@cli.com
Deepak Kapur	SUNY, Albany	(518)442-4281	kapur@albanycs.albany.edu
Dale Johnson	MITRE(Bedford)	(617)271-8894	
Andy Moore	NRL	(202)767-6698	moore@itd.nrl.navy.mil
Karl Levitt	UC Davis	(916)752-0832	levitt@ucdavis.edu
Sally Johnson	NASA, LaRC	(804)864-6204	scj@air16.larc.nasa.gov
Richard Platek	ORA	(607)277-2020	richard%oravax.uucp@cu-arpa.cs.cornell.edu
Bill Young	CLI	(512)322-9951	young@cli.com
Don Good	CLI	(512)322-9951	good@cli.com
Warren Hunt	CLI	(512)322-9951	hunt@cli.com
Jim Caldwell	NASA	(804)864-6214	jlc@air16.larc.nasa.gov, caldwell@cs.cornell.edu
Mark Bickford	ORA	(607)277-2020	??? %oravax.uucp@cu-arpa.cs.cornell.edu
Mark Saadtink	ORA	(613)238-7900	mark@ora.on.ca
Bill Legato	NSA	(301)688-4229	
Kang Shin	U. Michigan	(313)763-0391	kgshin@eecs.umich.edu
Chuck Meissner	NASA, LaRC	(804)864-6218	cwm@air12.larc.nasa.gov
Pete Saraceni	FAA Tech Ctr.	(609)484-5577	
Roger Kieckhafer	U. Nebraska	(402)472-2402	rogerk@fergvax.unl.edu
Doug Hoover	ORA	(607)277-2020	hoove%oravax.uucp@cu-arpa.cs.cornell.edu
David Guaspari	ORA	(607)277-2020	??? %oravax.uucp@cu-arpa.cs.cornell.edu
Tom Schubert	UC Davis	(916)752-6452	schubert@iris.ucdavis.edu
Mark Ardis	SEI	(412)268-7636	maa@sei.cmu.edu
Clive Pygott	RSRE, Malvern	44-684-895835	"PYGOTT%hermes.mod.uk"@relay.MOD.UK
John Kershaw	RSRE, Malvern	44-684-895845	"KERSHAW%hermes.mod.uk"@relay.MOD.UK
David Musser	RPI	(518)276-8660	musser@turing.cs.rpi.edu
Phil Windley	U. Idaho	(208)885-6501	windley@ted.mrc.uidaho.edu
John Knight	U. Va.	(804)982-2216	jck@neptune.cs.virginia.edu
Rick Kuhn	NIST	(301)975-3337	
Dave Eckhardt	NASA, LaRC	(804)864-1698	dee@csab.larc.nasa.gov
John McHugh	CLI	(919)493-4932	mchugh@cli.com
Milt Holt	NASA, LaRC	(804)864-1596	
Ben DiVito	Vigyan/NASA	(804)864-4883	bld@air16.larc.nasa.gov
Paul Miner	NASA, LaRC	(804)864-6201	psm@air16.larc.nasa.gov
Rick Butler	NASA, LaRC	(804)864-6198	rwb@air16.larc.nasa.gov
M.K. Srivas	ORA	(607)277-2020	srivas%oravax.uucp@cu-arpa.cs.cornell.edu
John Rushby	SRI	(415)859-5456	rushby@csl.sri.com
N. Shankar	SRI	???	shankar@csl.sri.com
Gerry Cohen	Boeing	(206)865-3739	NOT ON INTERNET

6 51-06 P-24  
319682

# **DIGITAL AVIONICS A CORNERSTONE OF AVIATION**

by  
**Cary R. Spitzer**  
**NASA Langley Research Center**

Presented to the NASA Formal Methods Workshop  
by  
**Charles W. Meissner, Jr.**

**N91-17560**



# **DIGITAL AVIONICS - A CORNERSTONE OF AVIATION**

**INTRODUCTION**

**CURRENT EXAMPLES**

**CURRENT ISSUES**

**FUTURE TRENDS**

**INTERNATIONAL SCENE**

**SUMMARY**

# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## INTRODUCTION: Avionics Roles

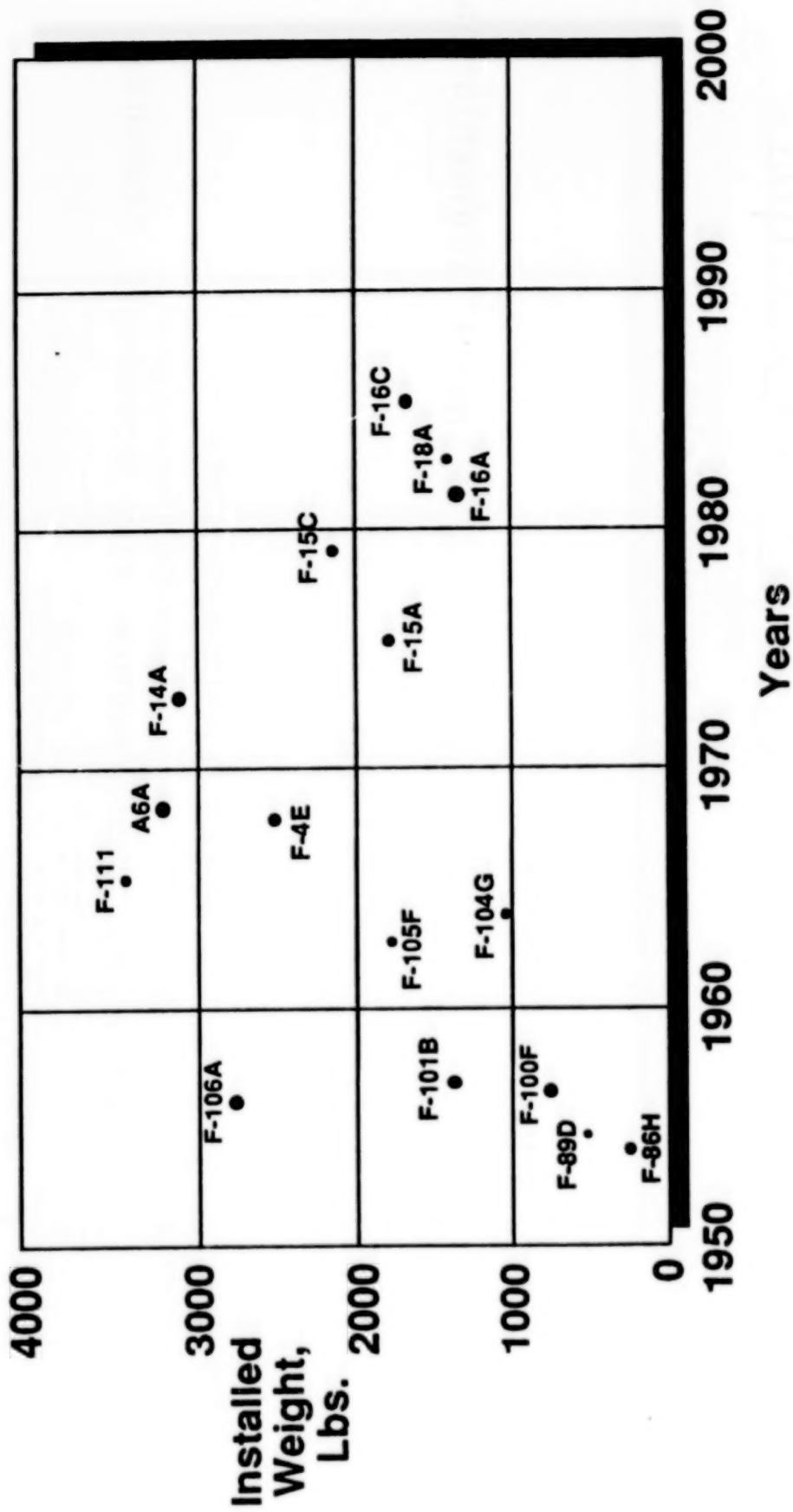
- Communication
  - HF and VHF
  - Satellite
  - Data Links
- Navigation
  - Ground-based systems
  - Inertial and satellite-based systems
- Goal: Autonomous operation!!!

# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

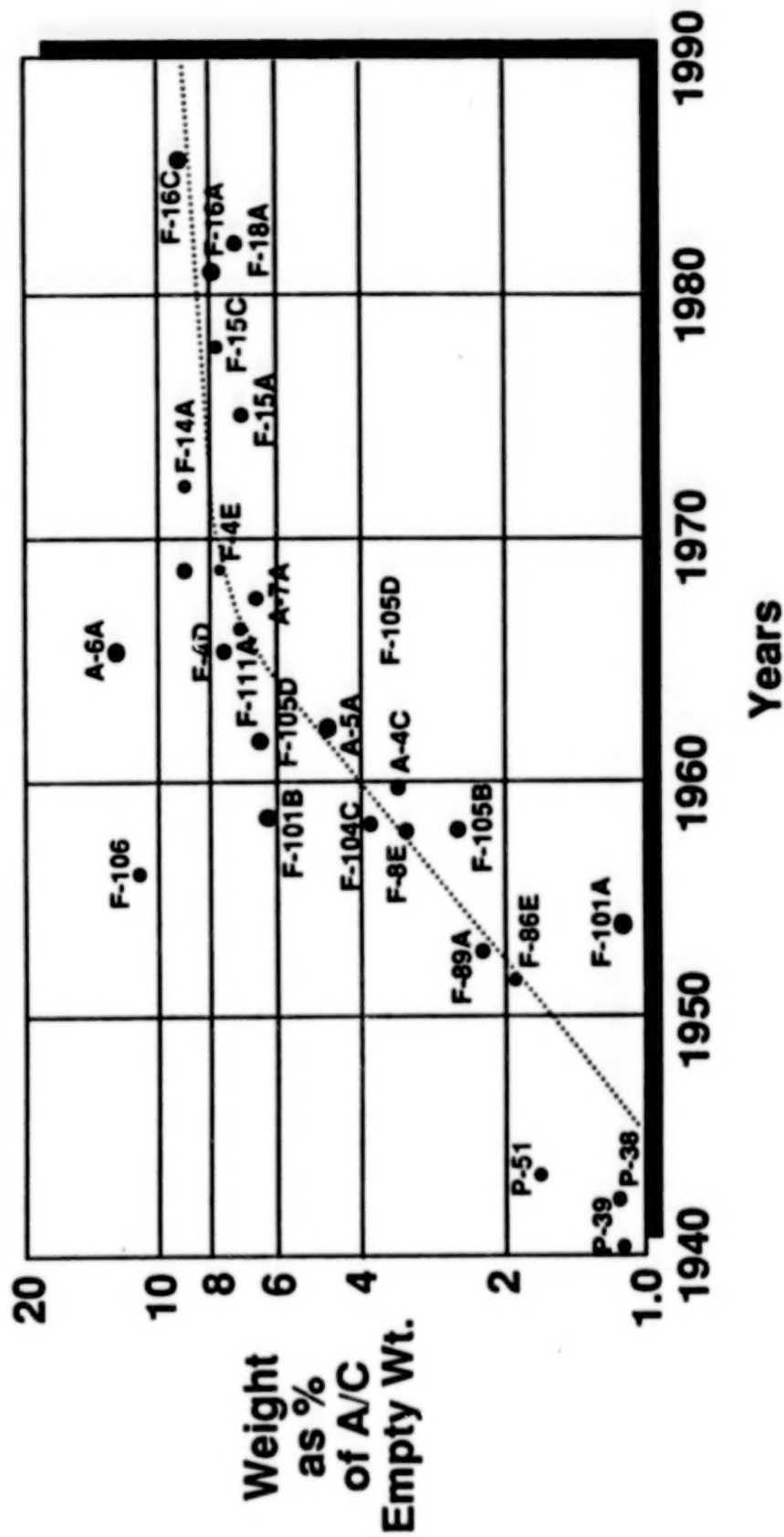
## INTRODUCTION: Avionics Roles

- Fly-by-wire flight controls
  - Historically used for stability & control augmentation
    - Not flight critical
- Emerging as a flight critical system
  - Driven by performance and economic demands
  - F-16, A-320, B-777

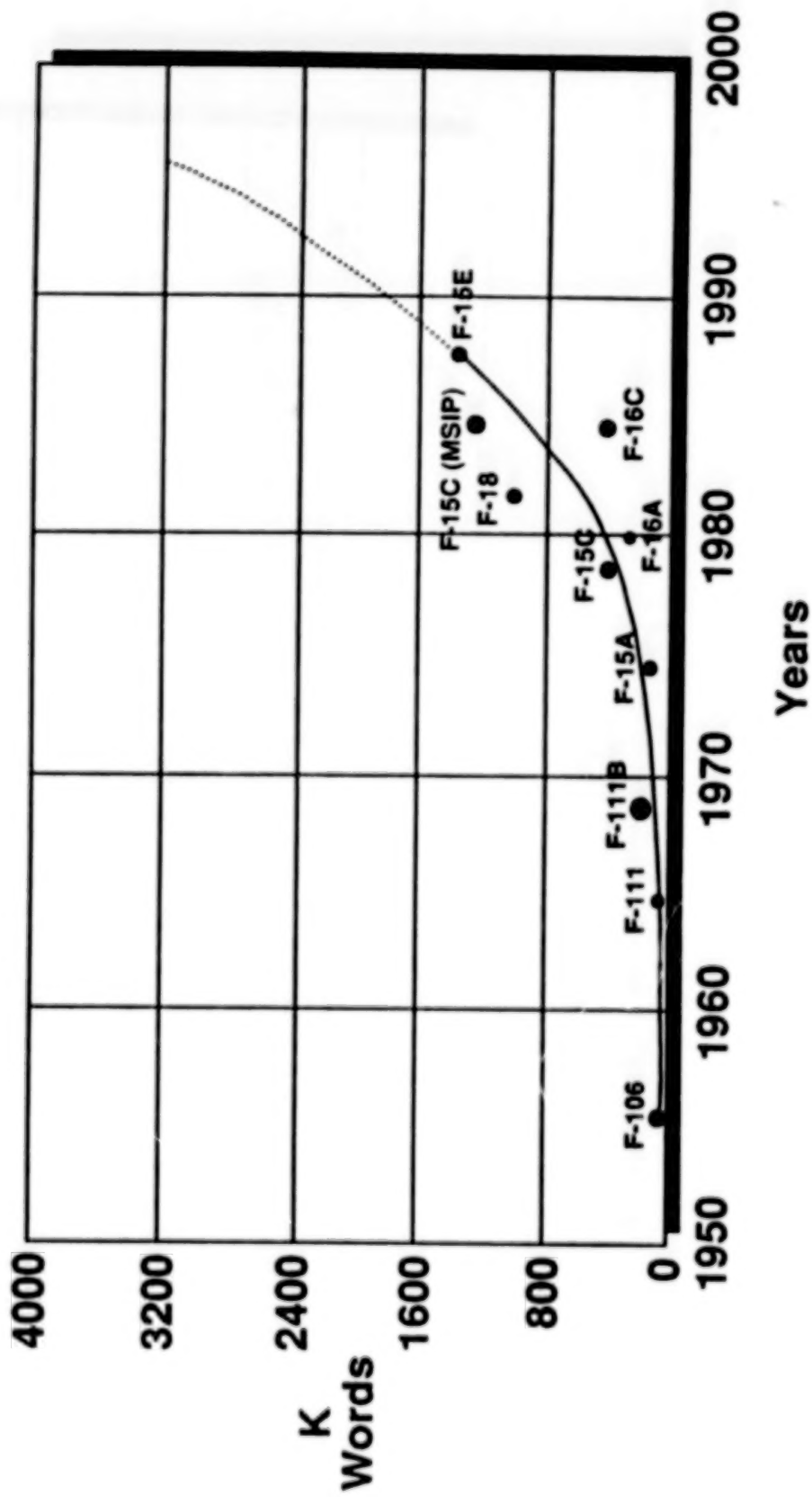
# **FIGHTER INSTALLED AVIONICS WEIGHT**



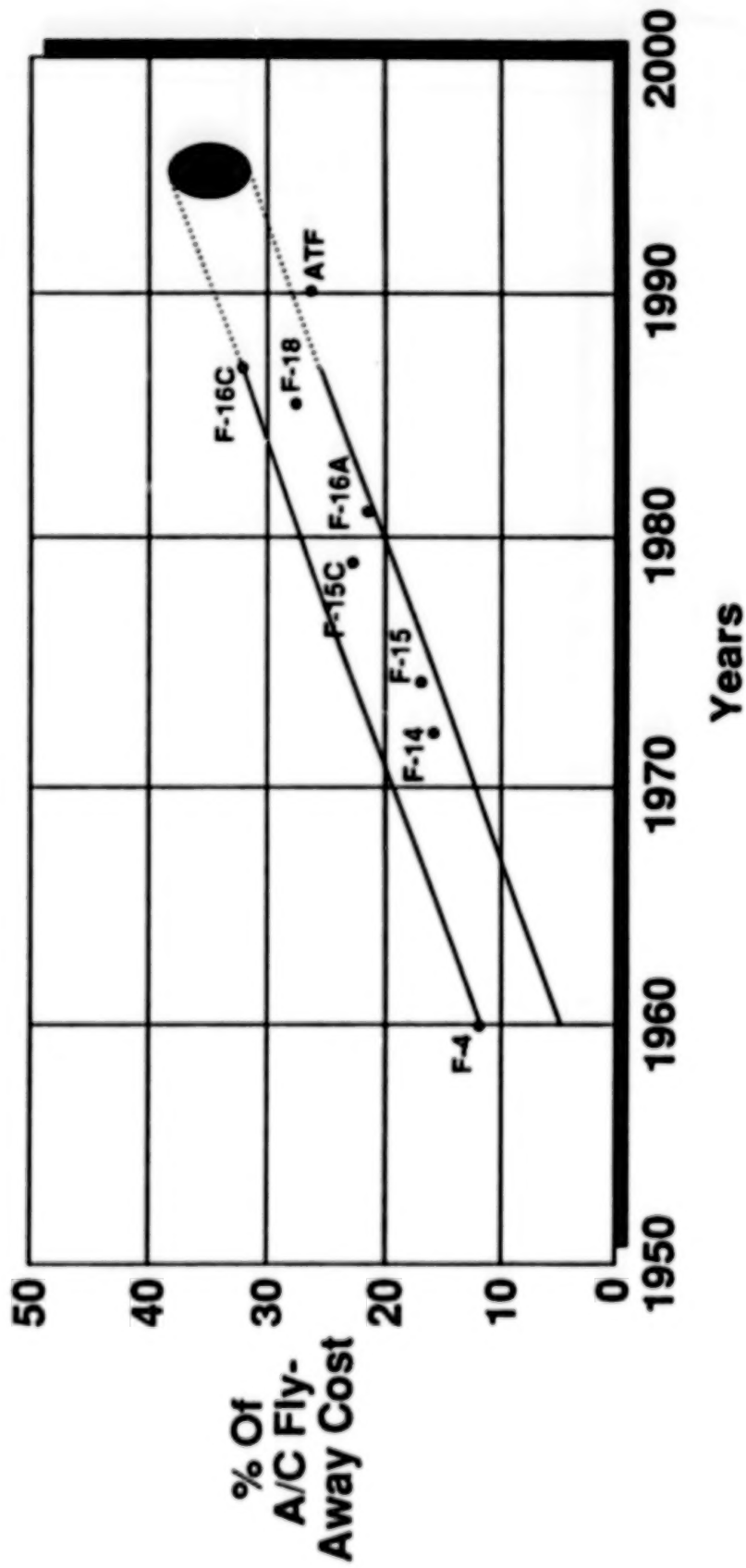
# TRENDS IN AVIONICS ABOARD FIGHTER/ATTACK AIRCRAFT



# TOTAL ON BOARD COMPUTER CAPACITY (OFP)

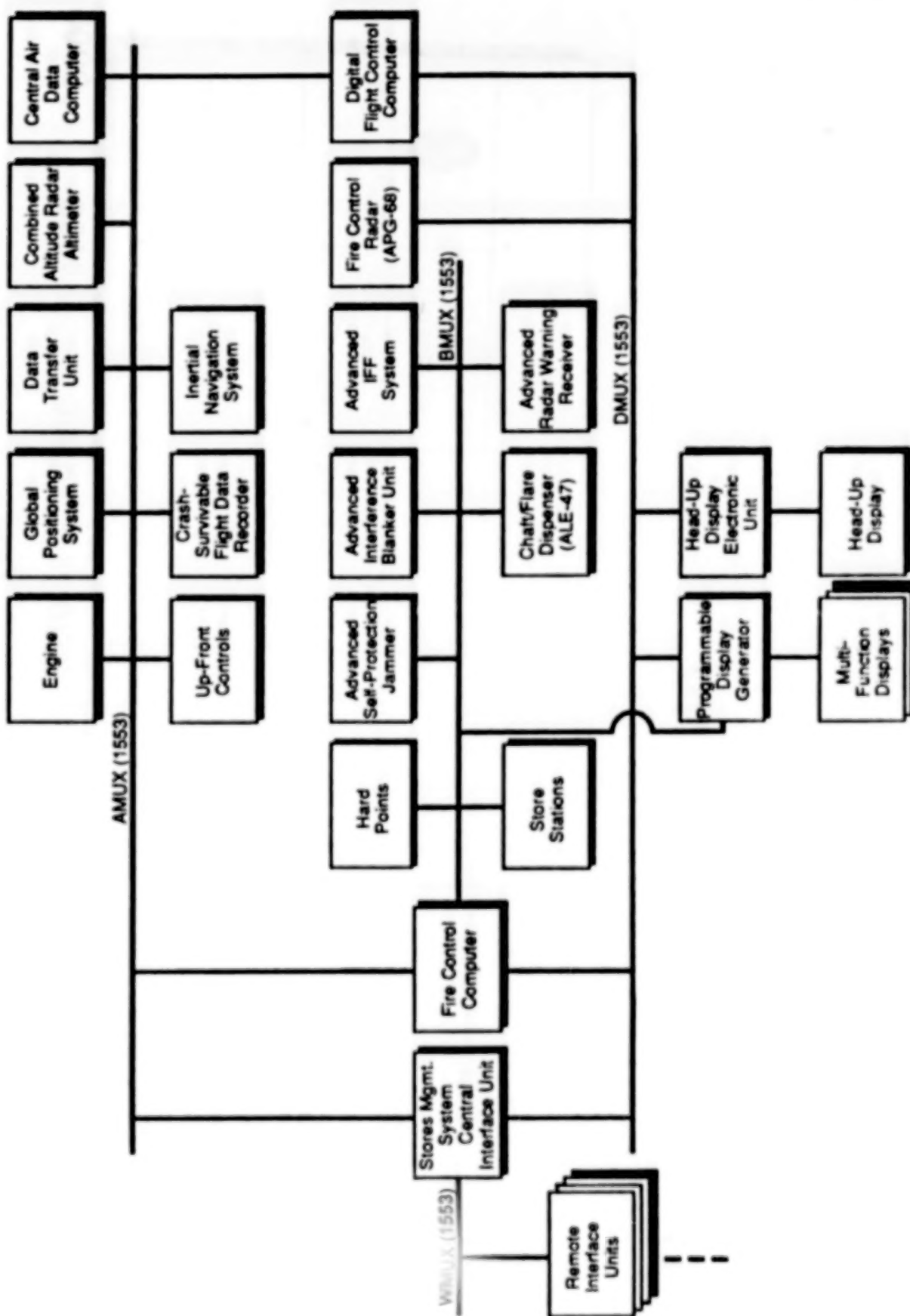


# AVIONICS COST TREND





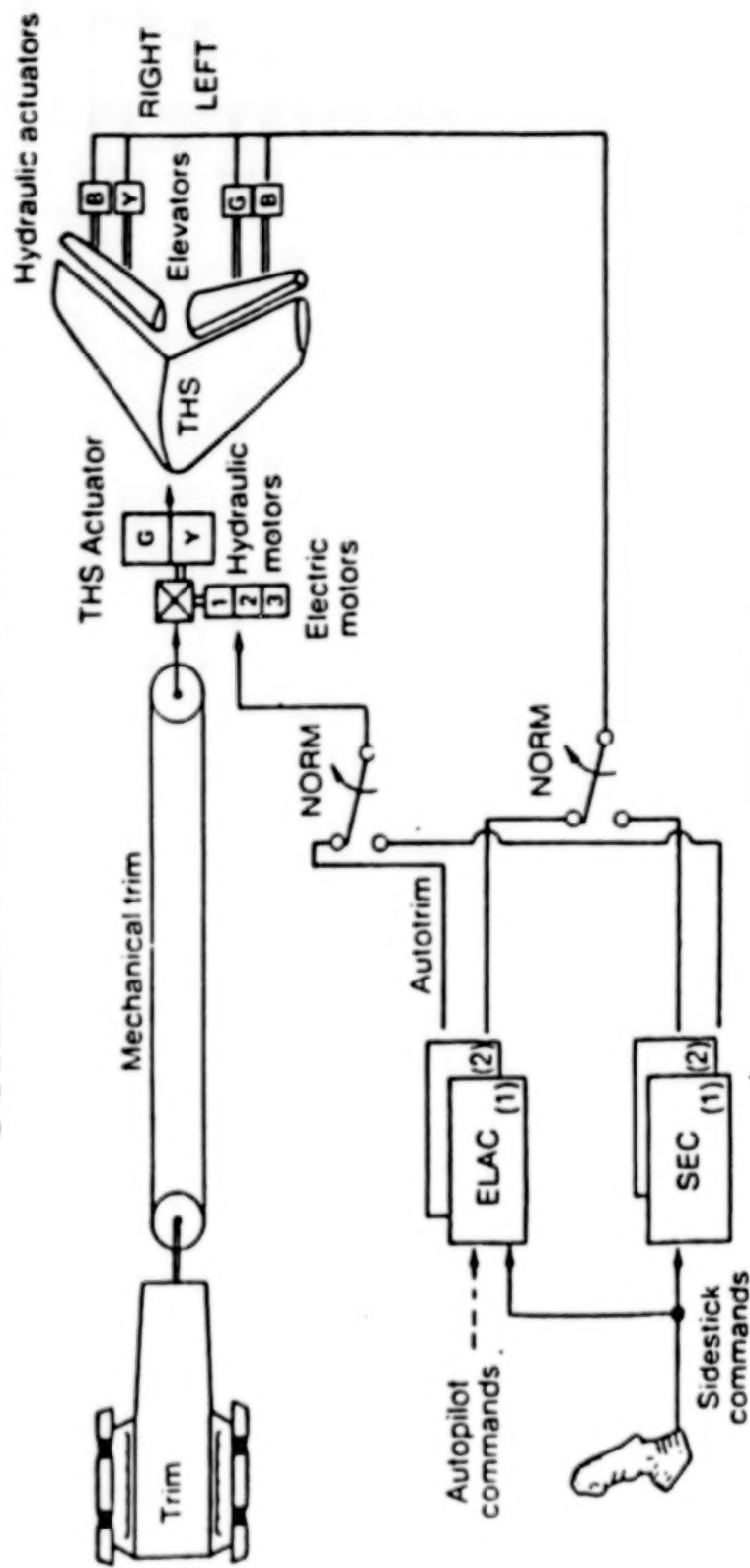
# F-16 AVIONICS SYSTEM ARCHITECTURE





# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## CURRENT EXAMPLES: A-320

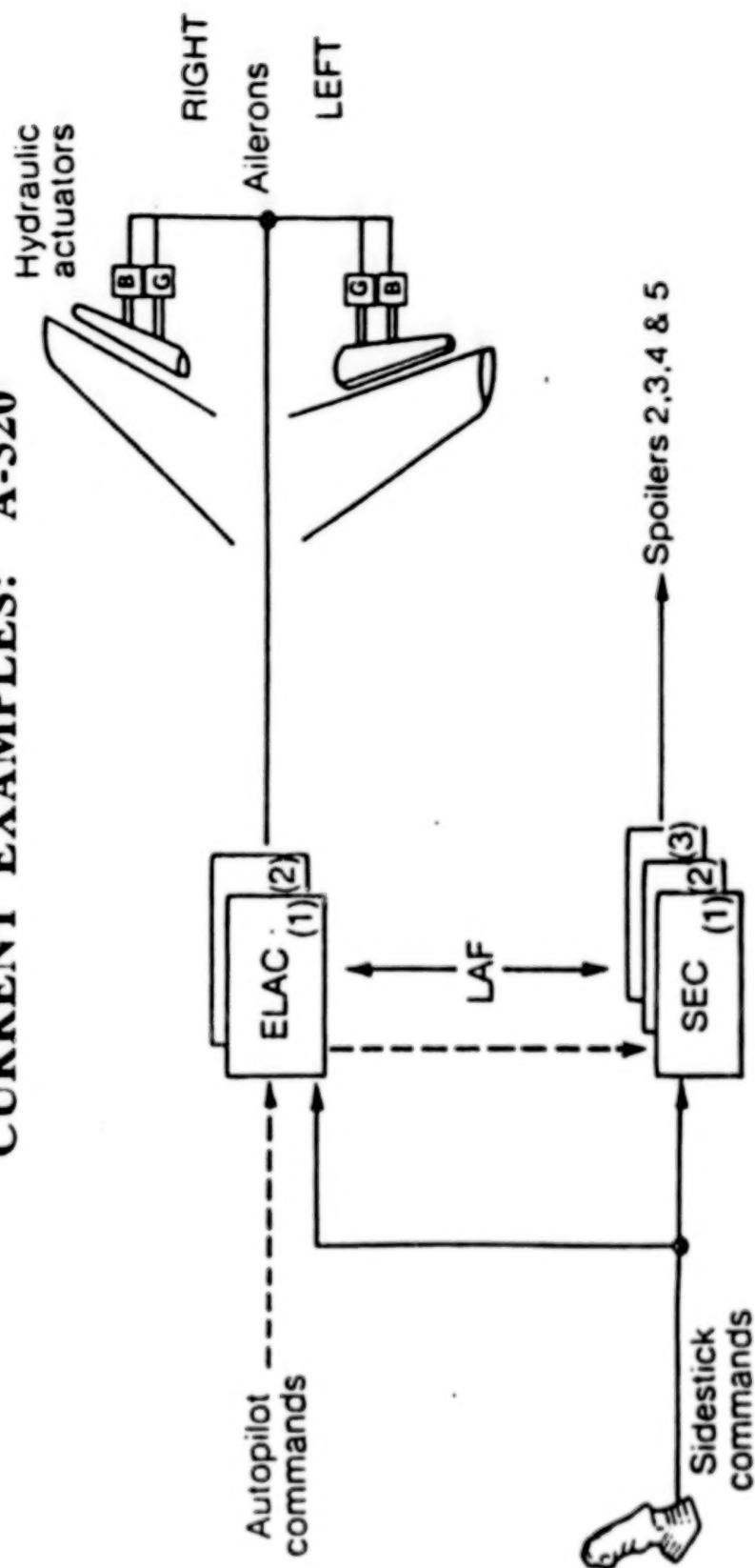


THS	Trimmable horizontal stabilizer	Hydraulic
ELAC	Elevator and aileron computer	B Blue system
SEC	Spoiler and elevator computer	G Green system
		Y Yellow system

## Pitch Control

# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## CURRENT EXAMPLES: A-320

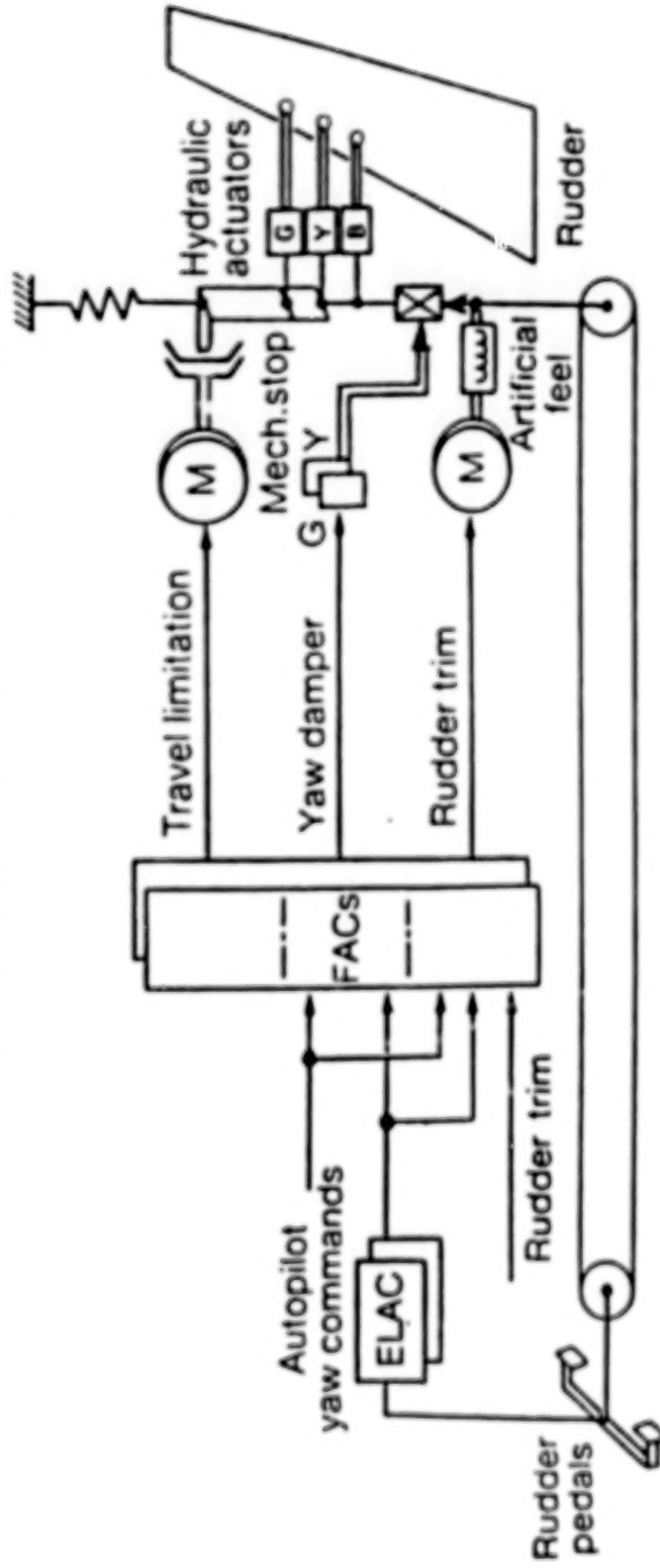


LAF	Load alleviation function	Hydraulic
ELAC	Elevator and aileron computer	B Blue system
SEC	Spoiler and elevator computer	G Green system
		Y Yellow system

## Roll Control

# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

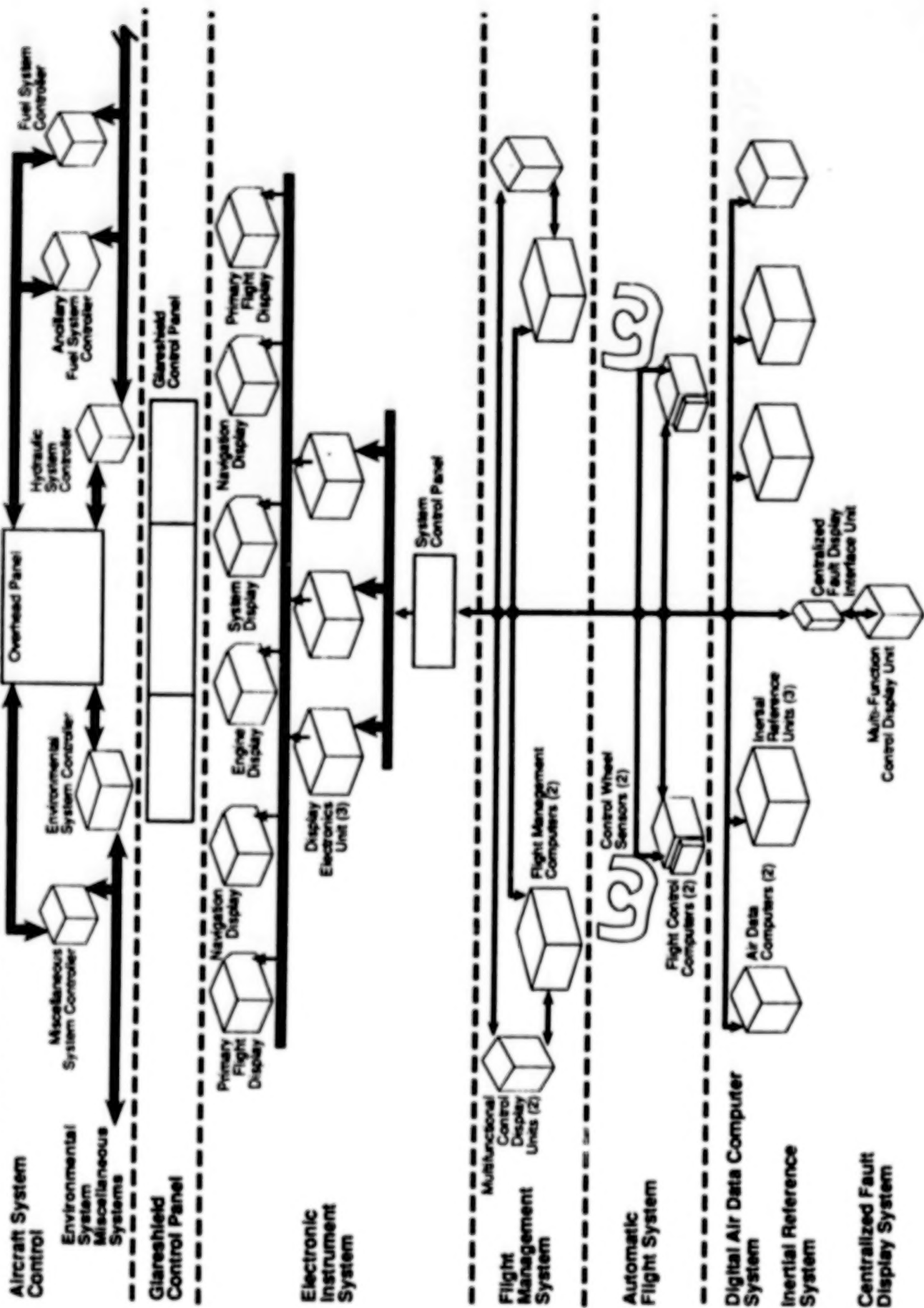
## CURRENT EXAMPLES: A-320



M	Motor actuator	Hydraulic
FAC	Flight augmentation computer	Blue system
		Green system
		Yellow system

## Yaw Control

# MD-11 FLIGHT GUIDANCE / FLIGHT DECK SYSTEM



## **DIGITAL AVIONICS - A CORNERSTONE OF AVIATION**

### **CURRENT ISSUES: Hardware**

- Modeling of complex systems
  - Proof of fault tolerance, high reliability
- Electromagnetic interference
  - Growing concern due to composite aircraft, increased emission of RF, and smaller electronic element sizes



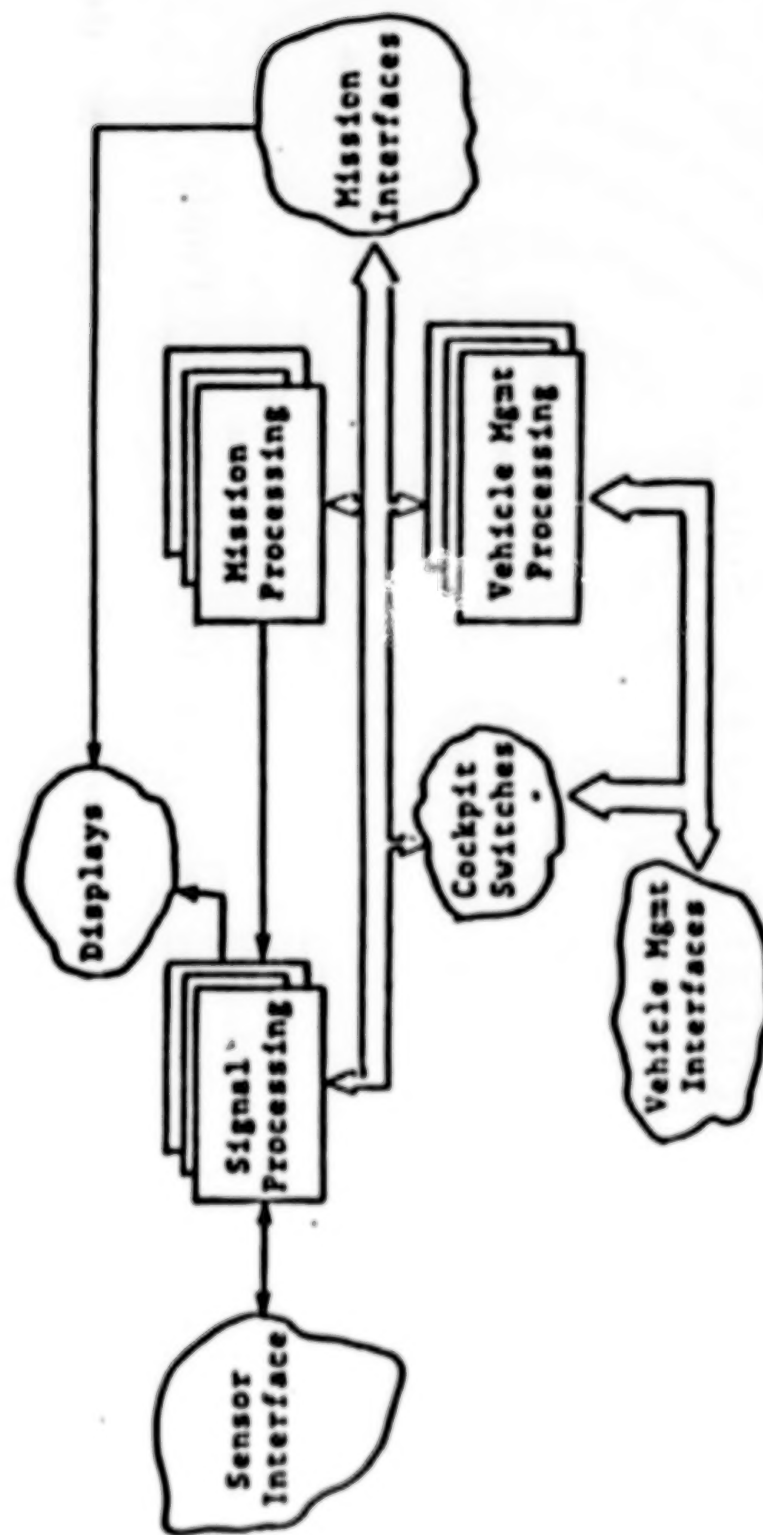
## DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

### **CURRENT ISSUES: Software**

- Developing competency in Ada
  - Mandated for DoD, Space Station Freedom, civil transports
- Computer-Aided Software Engineering (CASE) Tools
  - Capabilities for real-time software analysis & design
  - Tool validation

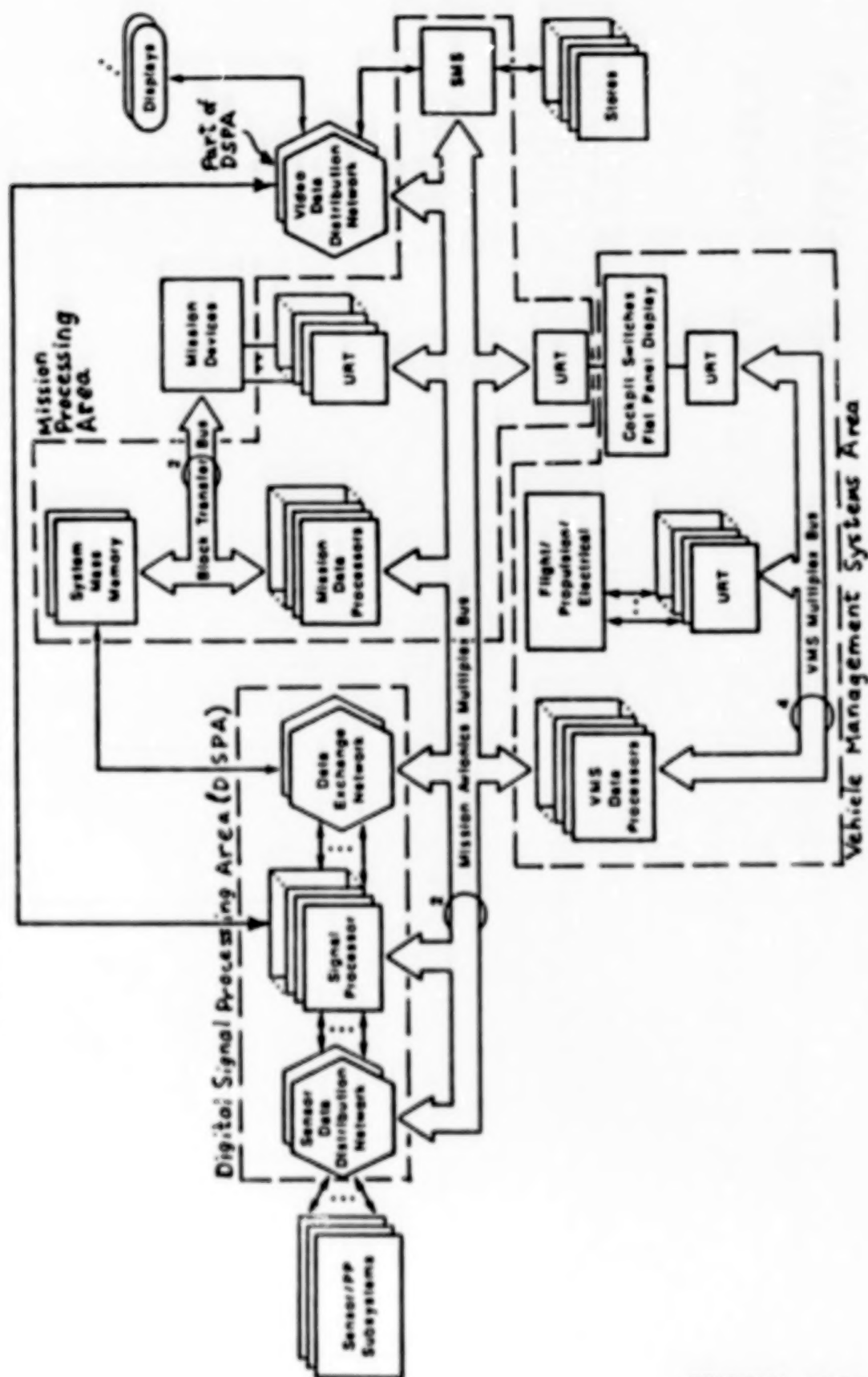
# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## FUTURE TRENDS: PAVE PILLAR



# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## FUTURE TRENDS: PAVE PILLAR

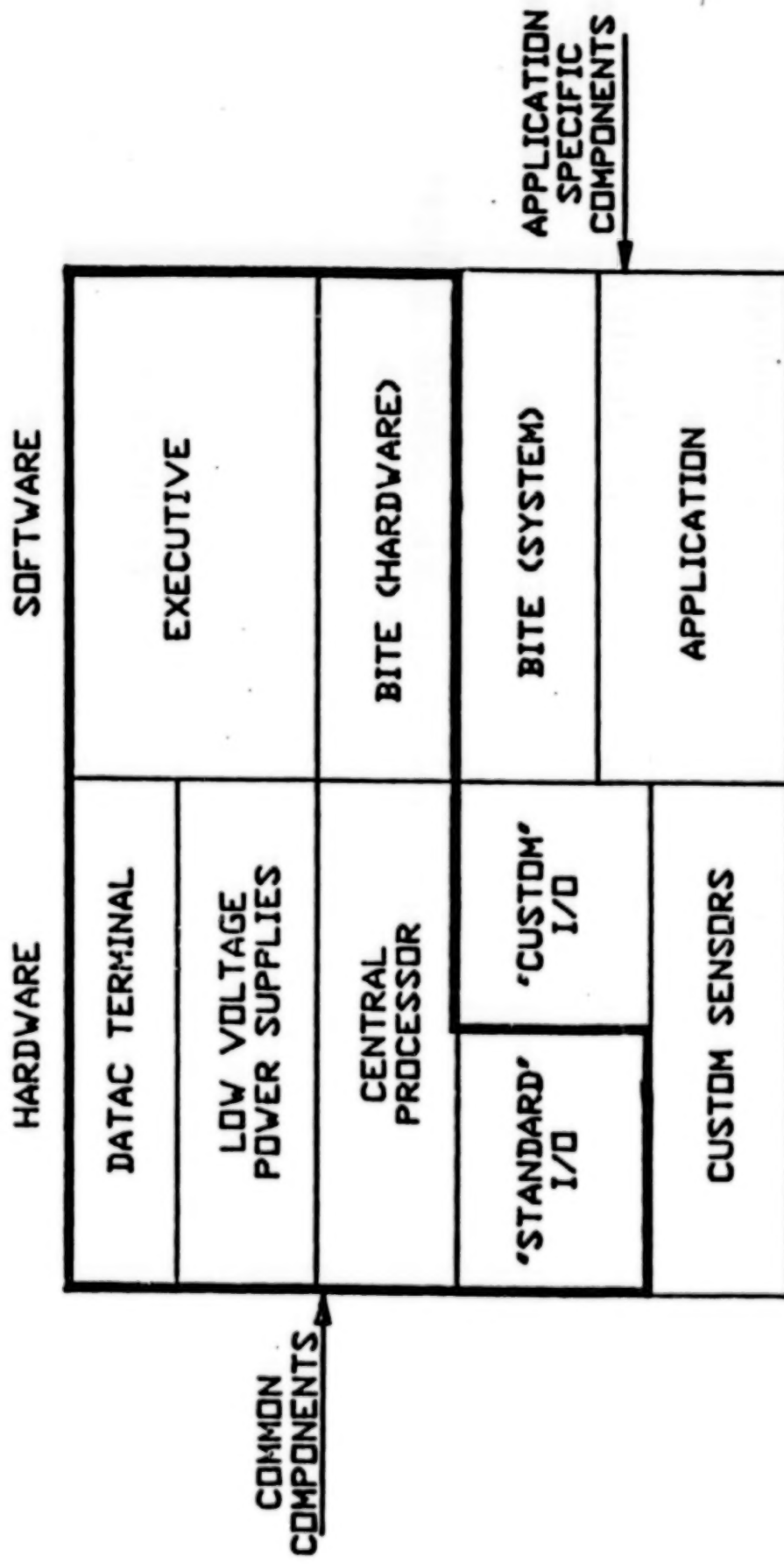


ORIGINAL PAGE IS  
OF POOR QUALITY



# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## FUTURE TRENDS: INTEGRATED MODULAR AVIONICS



COMPONENTS OF TYPICAL LRU

## **DIGITAL AVIONICS - A CORNERSTONE OF AVIATION**

### **FUTURE TRENDS: Supporting Technologies**

- Flat panel, full color, liquid crystal displays
- Replacing CRTs
- Advanced formats; not electronic steam gauges
- Higher speed data buses
- Artificial intelligence pioneer programs
- Faultfinder
- Diverter
- Pilot's Associate

## DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

### INTERNATIONAL SCENE: Europe

- Leading firms are GEC Avionics, Smiths Industries, Sextant, & Aerospatiale

- Extremely capable; serious competition for U.S. firms

- Build most of the Airbus avionics

- GEC Avionics will build the B-777 flight control system

- Build flight controls for Jaguar and YC-14

- European Community 92 will strengthen competitive threat



## DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

### INTERNATIONAL SCENE: Japan

- An emerging competitor in the world market
- Historically has been component oriented: displays, microprocessors, etc.
- Lack system design and analysis, & software capabilities
  - FS-X program will help to build a foundation for military & civil avionics
- MITI has established a committee to define an avionics technology development plan

# DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

## SUMMARY

- Continually expanding role for avionics
- Flight critical avionics are here
- Strong emphasis on Ada
- Module-based architectures emerging
- Artificial intelligence applications being developed
- Significant competitive threat to U.S. firms from Europe & Japan

30  
52-08 P-22  
519685

**LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

**JIM McWHA**

**CHIEF ENGINEER - FLIGHT CONTROLS  
BOEING COMMERCIAL AIRPLANE GROUP**

**N91-17561**

**AUGUST 20, 1990**

## LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

### INDUSTRY STATUS

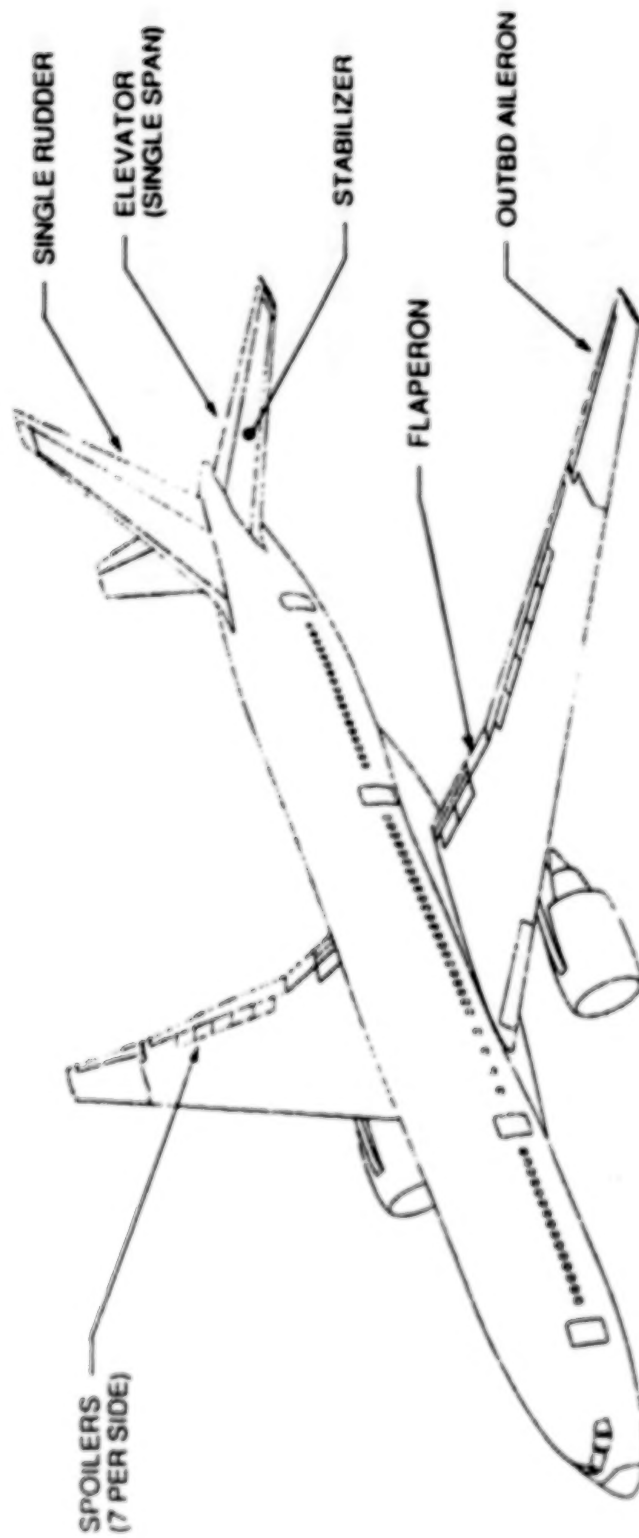
DIGITAL AUTOPILOT SYSTEMS WERE FIRST CERTIFICATED FOR USE ON COMMERCIAL AIRPLANES IN THE LATE 1970'S

THE A-320 AIRPLANE WAS THE FIRST COMMERCIAL AIR TRANSPORT AIRPLANE TO BE CERTIFICATED WITH A FLY BY WIRE PRIMARY FLIGHT CONTROL SYSTEM

BOEING WILL HAVE ALL FLY BY WIRE FLIGHT CONTROLS ON THE 767-X (777) AIRPLANE

## **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

- o DEFINITION**
- o SAFETY**
- o INDUSTRY STATUS**
- o PROGRAM PHASES**



**767-X PRIMARY FLIGHT CONTROL SURFACES**

## LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

### DEFINITION

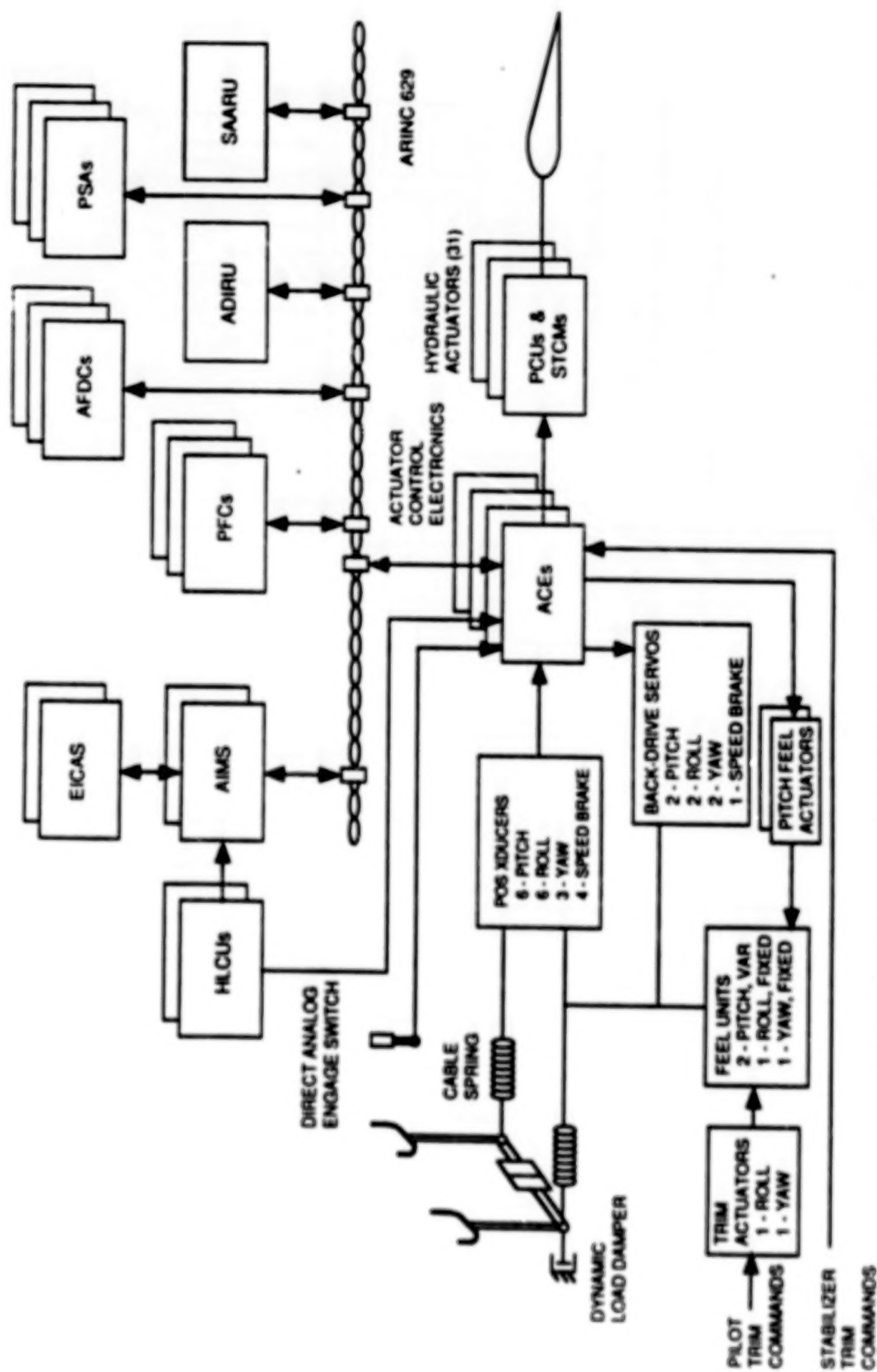
A CONTROL SYSTEM IMPLEMENTED IN DIGITAL COMPUTER TECHNOLOGY WHICH HAS A FUNCTION WHICH IF NOT PERFORMED AS INTENDED IS LIFE THREATENING

EXAMPLES: AN AUTOPILOT USED FOR AUTOMATIC LANDING IN LOW  
VISIBILITY CONDITIONS

AN AIRPLANE CONTROL SYSTEM IMPLEMENTED WITHOUT  
CONTROL CABLES:

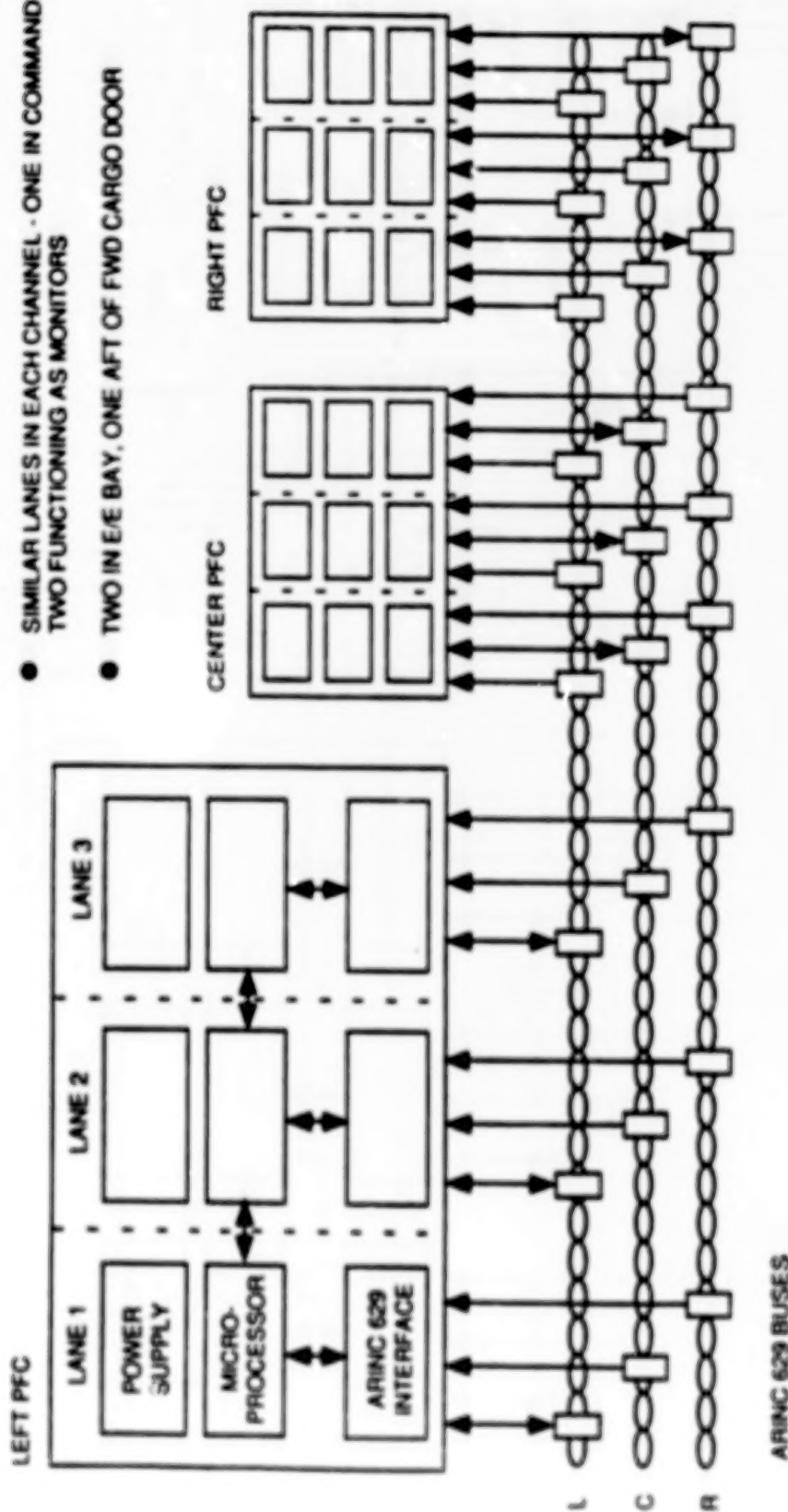
FLY BY WIRE

FLY BY LIGHT



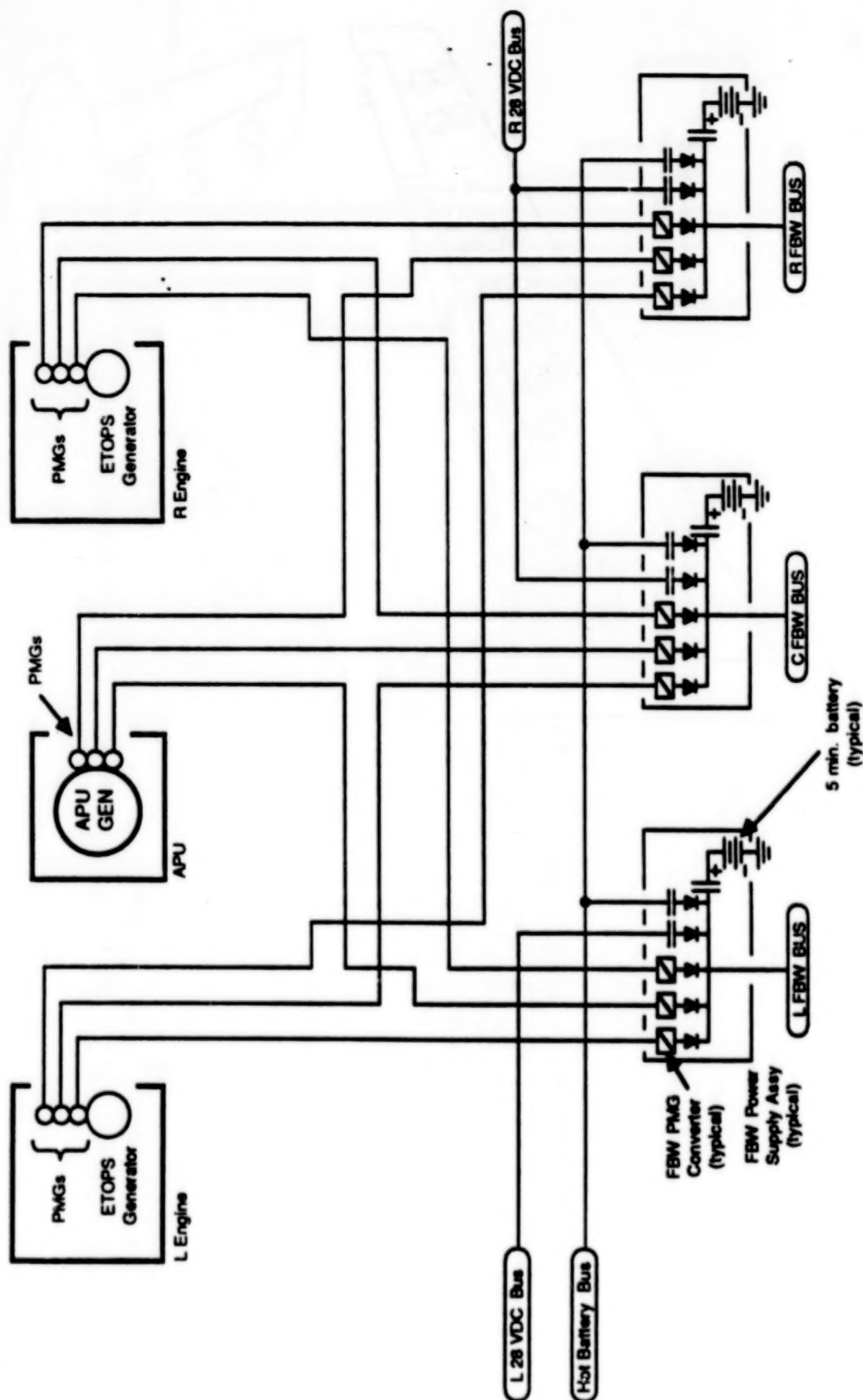


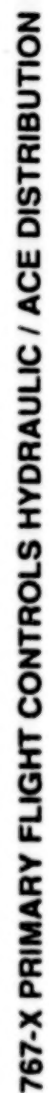
- DISSIMILAR CHANNELS - LEFT, CENTER, RIGHT
- SIMILAR LANES IN EACH CHANNEL - ONE IN COMMAND, TWO FUNCTIONING AS MONITORS
- TWO IN E/E BAY, ONE AFT OF FWD CARGO DOOR



## PRIMARY FLIGHT COMPUTER ARCHITECTURE

# 767-X ELECTRICAL POWER SYSTEM FLIGHT CRITICAL DC





## **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

### **SAFETY**

**FEDERAL AVIATION ADMINISTRATION (FAA) REGULATIONS DEFINE THE BASIC SAFETY**

#### **CRITERIA:**

**FAR 25.1309 NO SINGLE FAILURE OR COMBINATION OF FAILURES WHICH ARE NOT  
SHOWN TO BE EXTREMELY IMPROBABLE SHALL PREVENT  
CONTINUED SAFE FLIGHT AND LANDING OF THE AIRPLANE**

**EXTREMELY IMPROBABLE - PROBABILITY OF  $1 \times 10^{-9}$  OR LESS PER FLIGHT HOUR OR  
EVENT**

## **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

### **SAFETY**

FEDERAL AVIATION ADMINISTRATION (FAA) REGULATIONS DEFINE THE BASIC SAFETY

#### **CRITERIA:**

**FAR 25.1309 NO SINGLE FAILURE OR COMBINATION OF FAILURES WHICH ARE NOT  
SHOWN TO BE EXTREMELY IMPROBABLE SHALL PREVENT  
CONTINUED SAFE FLIGHT AND LANDING OF THE AIRPLANE**

**EXTREMELY IMPROBABLE - PROBABILITY OF  $1 \times 10^{-9}$  OR LESS PER FLIGHT HOUR OR  
EVENT**

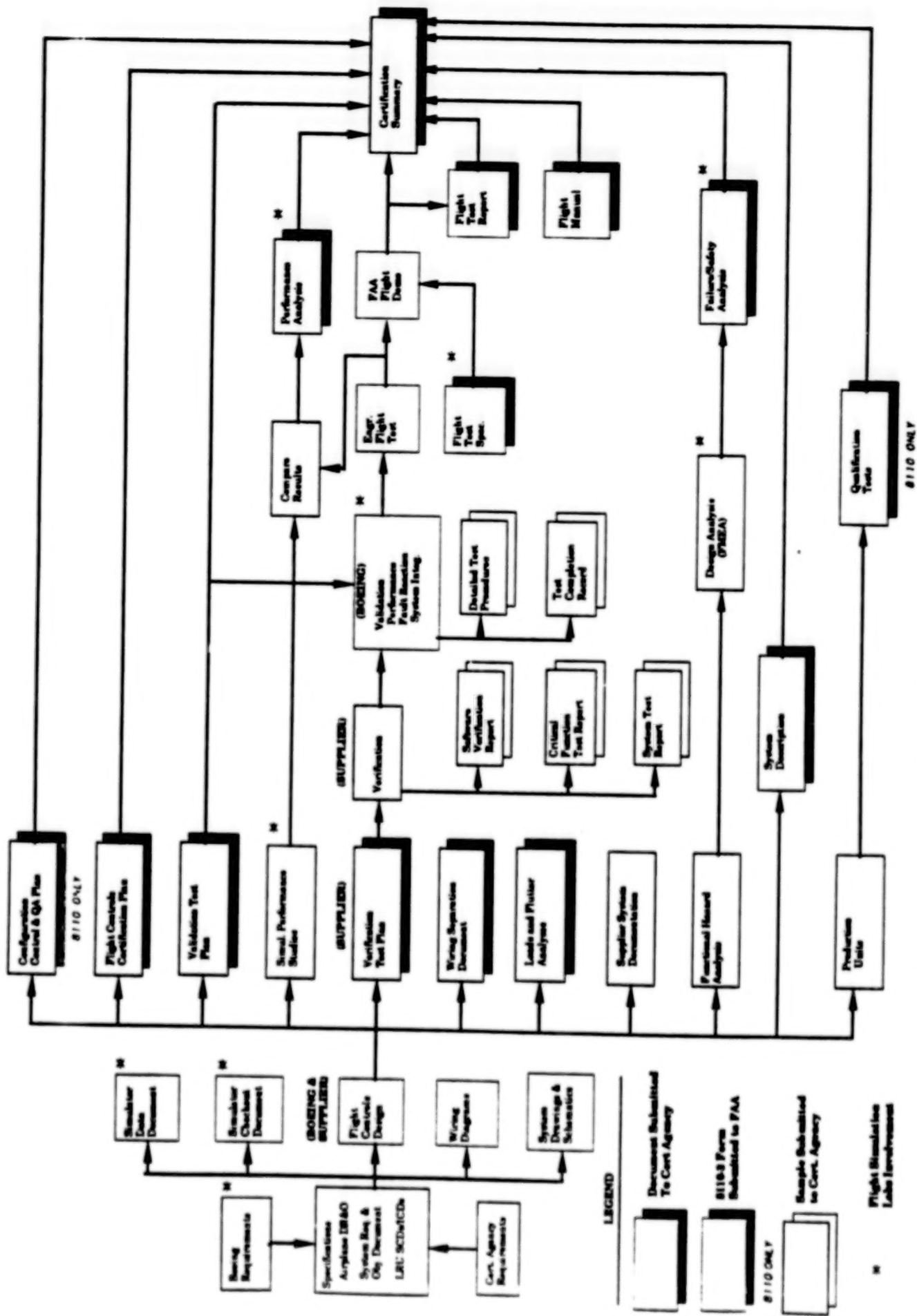


FIGURE 1.2-1 PRIMARY FLIGHT CONTROLS DEVELOPMENT OVERVIEW

# **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - REQUIREMENTS DEFINITION**

### **TOP DOWN STRUCTURED PROCESS:**

**AIRPLANE LEVEL REQUIREMENTS    TOP LEVEL DESIGN REQUIREMENTS AND  
OBJECTIVES**

### **SYSTEM REQUIREMENTS**

**CERTIFICATION REQUIREMENTS  
FUNCTIONAL REQUIREMENTS  
INTEGRITY REQUIREMENTS  
ARCHITECTURAL CONSIDERATIONS**

### **SOFTWARE REQUIREMENTS**

**EXPANSION OF SYSTEM REQUIREMENTS TO A  
LEVEL WHICH CAN BE IMPLEMENTED IN A TARGET  
DIGITAL COMPUTER OR COMPUTERS**



# **IFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - REQUIREMENTS DEFINITION (CONT)**

**DETAILED DESIGN  
REQUIREMENTS**

**REQUIREMENTS WHICH EVOLVE OUT OF THE USE OF  
SPECIFIC HARDWARE/SOFTWARE TO IMPLEMENT THE  
SOFTWARE REQUIREMENTS**

# **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - DESIGN AND DEVELOPMENT**

### **HARDWARE SELECTION**

**I/O REQUIREMENTS**

**PROCESSING SPEED**

**MEMORY SIZE**

**ETC**

### **PROGRAMMING LANGUAGE**

**INDUSTRY/COMPANY STANDARD**

**SUPPORT SOFTWARE AVAILABILITY AND MATURITY**

**LONG TERM MAINTENANCE**

**ETC**

### **CODE GENERATION**

**TYPICALLY AN INCREMENTAL BUILD PROCESS**

### **TESTING**

**HARDWARE - QUALIFICATION TESTING - RTCA DO-160**

**INCREMENTAL SOFTWARE LOADS - VENDOR AND AIRFRAME**

**SYSTEMS INTEGRATION / IRON BIRD**

**AIRPLANE - GROUND AND FLIGHT**

# **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - VERIFICATION**

**GUIDELINE DOCUMENT      RTCA DOCUMENT DO-178A**

**VERIFICATION PROCESSES ARE A FUNCTION OF SYSTEM CRITICALITY**

<b>CRITICAL SYSTEM</b>	<b>A FORMAL PROCESS OF ASSURING THAT ALL SOFTWARE REQUIREMENTS HAVE BEEN IMPLEMENTED <u>COMPLETELY</u> AND <u>EXCLUSIVELY</u></b>
------------------------	---

# **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - VALIDATION**

**A PROCESS OF ASSURING THAT ALL SYSTEM REQUIREMENTS HAVE BEEN  
IMPLEMENTED CORRECTLY**

### **o ANALYSES**

#### **SAFETY ANALYSIS**

**HAZARD ASSESSMENT AND FAILURE  
ANALYSIS TO ASSURE THAT REQUIREMENTS  
OF FAR 25.1309 ARE SATISFIED**

#### **PERFORMANCE ANALYSIS**

**ASSURANCE THAT SYSTEM PERFORMS  
INTENDED FUNCTION WITHIN ACCEPTABLE  
LIMITS UNDER ALL ALLOWABLE  
ENVIRONMENTAL AND TOLERANCE  
CONDITIONS**

## LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

### PROGRAM PHASES - VALIDATION (CONT)

- o LABORATORY TESTING      TEST TO ISOLATE ERRORS AND PROBLEMS  
BEFORE FLIGHT TEST. TEST UNDER NORMAL AND  
FAILURE CONDITIONS
- o SYSTEMS INTEGRATION      TEST WITH AS MANY INTERFACING SYSTEMS AS  
TESTING                      POSSIBLE TO ENSURE COMPATIBILITY
- o AIRPLANE GROUND              CHECK OF SYSTEMS INSTALLED IN AN AIRPLANE  
TESTING                      INCLUDING EMI/HIRF TESTS
- o AIRPLANE FLIGHT              COMPREHENSIVE TEST OF PERFORMANCE IN  
TESTING                      FLIGHT UNDER A VARIETY OF CONDITIONS USED  
TO CROSS CHECK SIMULATION RESULTS -  
AUTOLAND SYSTEM COULD REQUIRE 200-300  
LANDINGS OVER AN 8 MONTH PERIOD

## **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

### **PROGRAM PHASES - CERTIFICATION**

THE PROCESS OF DEMONSTRATING TO THE REGULATORY AUTHORITIES THAT ALL SAFETY AND PERFORMANCE REQUIREMENTS ARE SATISFIED

STARTS WITH A CERTIFICATION PLAN WHICH:

IDENTIFIES REGULATIONS AND ACCEPTABLE MEANS OF COMPLIANCE METHODS

DESCRIBES PROPOSED METHODS OF ESTABLISHING COMPLIANCE

DESCRIBES THE METHODS AND PROCESSES TO BE USED TO ASSURE AN ORDERLY AND CONTROLLED DESIGN AND DEVELOPMENT PROCESS

FOLLOW ON SPECIALIST MEETINGS

PERFORMANCE AND INTEGRITY DEMONSTRATIONS

# **LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS**

## **PROGRAM PHASES - CERTIFICATION (CONT)**

### **CERTIFICATION SUMMARY**



**CONFIRMS COMPLETE IMPLEMENTATION OF THE PROCESSES IDENTIFIED IN THE  
CERTIFICATION PLAN**

**PROVIDES A MEANS FOR ESTABLISHING VERIFICATION AND VALIDATION  
COVERAGE**





# 767-X PFCS Schedule





1990	1991	1992	1993	1994	1995
------	------	------	------	------	------

Type Cert. Appl.  Type Cert. 

767-X

FAA Interim Type Board  FAA Interim Type Board 


Rollout 

Cert Plan 4/1  FAA Cert Plan Approval 11/29  Fail/Safety System Des.  Final Cert. Data 

Certification Documentation

PRE-PRODUCTION DEVELOPMENT	Iron Bird Test	757 Flight Test
----------------------------	----------------	-----------------

757 PFCS Testing

Production Design Validation 

Build 767-X Units	Iron Bird Test	767-X FLIGHT TEST
-------------------	----------------	-------------------

767-X PFCS

52  
53-05 P-42  
219624

# Advanced Embedded Processing Present and Future

N91-17562

The Boeing Company  
G.C. Cohen

Boeing Military Airplanes

**Integrated Airframe/Propulsion Control System Architecture  
(IAPSA II)**

**Began:**  
**Ended:**

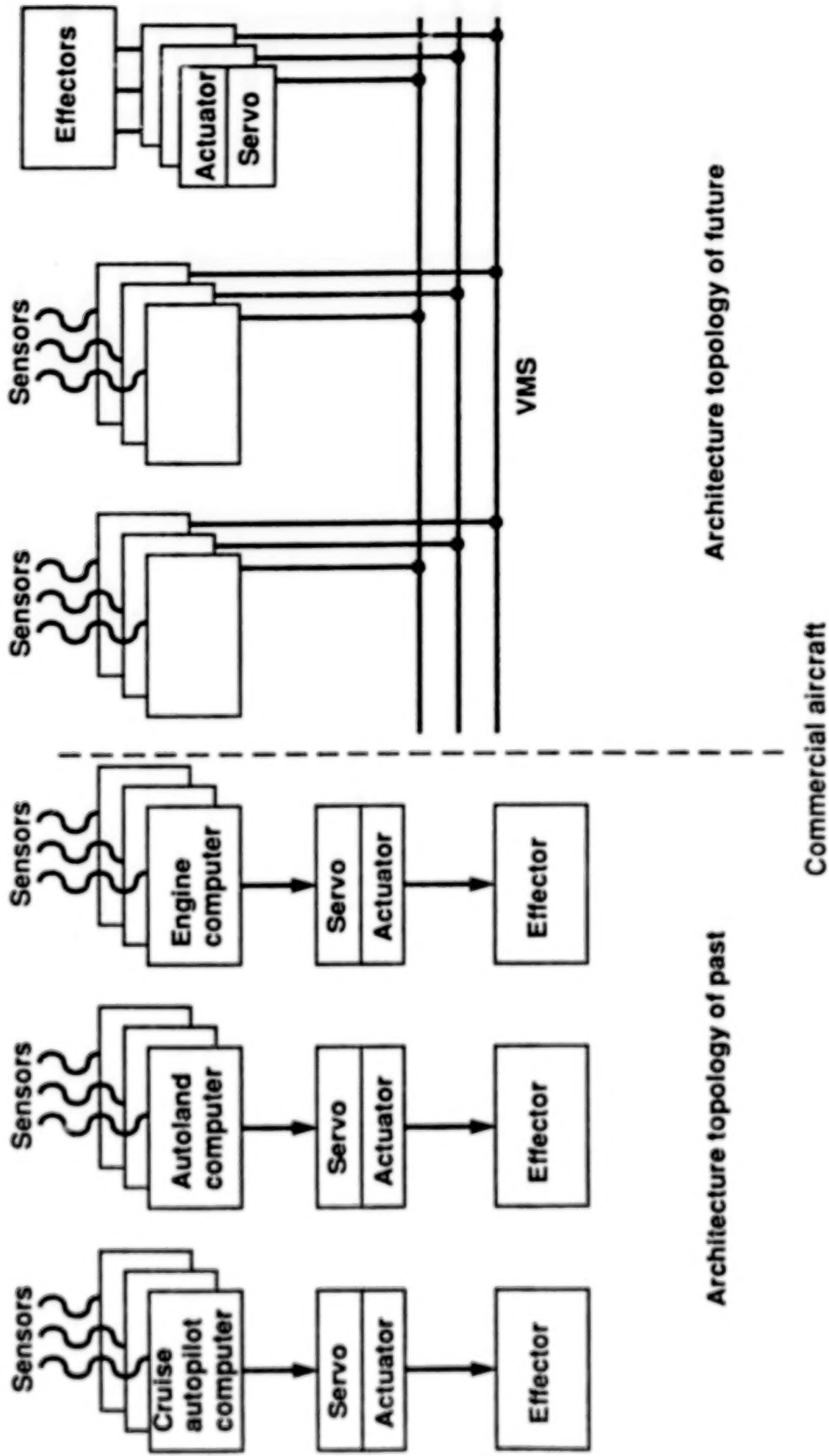
**July 26, 1985**  
**April 1, 1990**

## Goals of the Program

1. Design and validation methodology for systems architecture
2. Critical validation issues simulated in Airlab
3. System design
4. System specification
5. Small-scale system testing

# Methodology

# Why a Methodology



## Problem

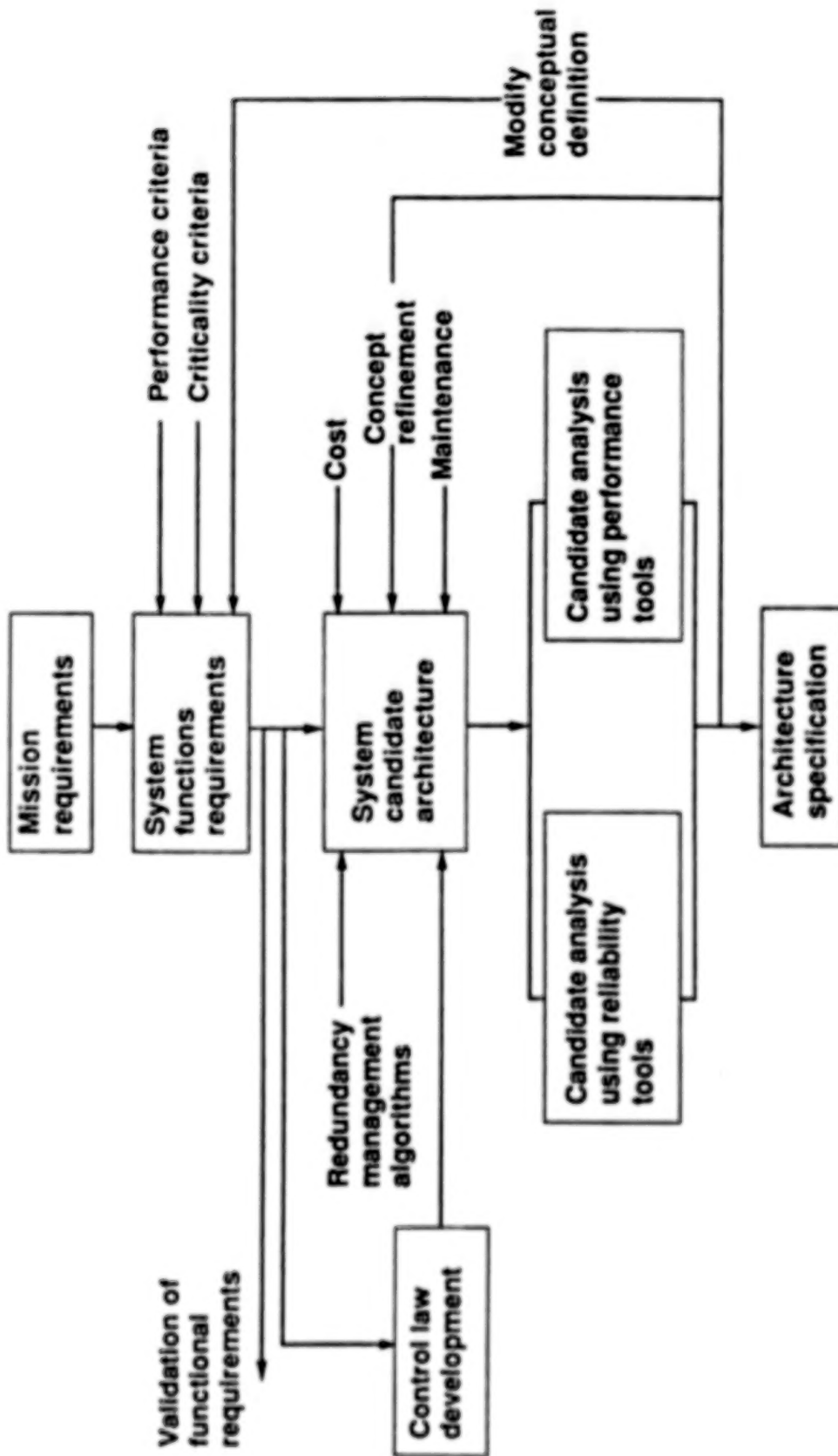
- Interrelationships difficult to specify in terms of meaningful requirements
  - Normal mode
  - Failure mode
- Unless contractor/vendor team takes a systems approach, system will be overdesigned and still may not meet the requirements



## Methodology Elements

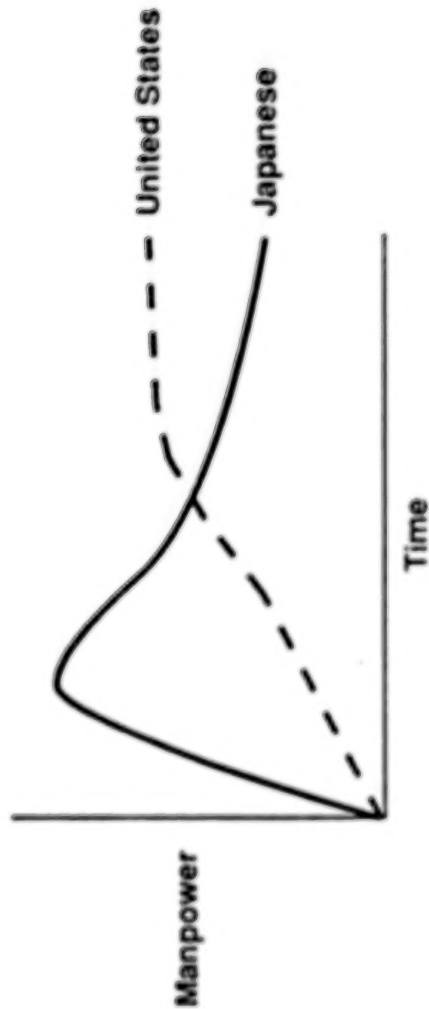
- Requirements
- Specifications
- Design
  - Automation
  - Guidelines
  - Building blocks
  - Reliability
    - Availability
  - Performance
    - Survivability
  - Design for validation
    - Maintainability
  - Design for cost
  - Proof of correctness
- Testing
- Traceability

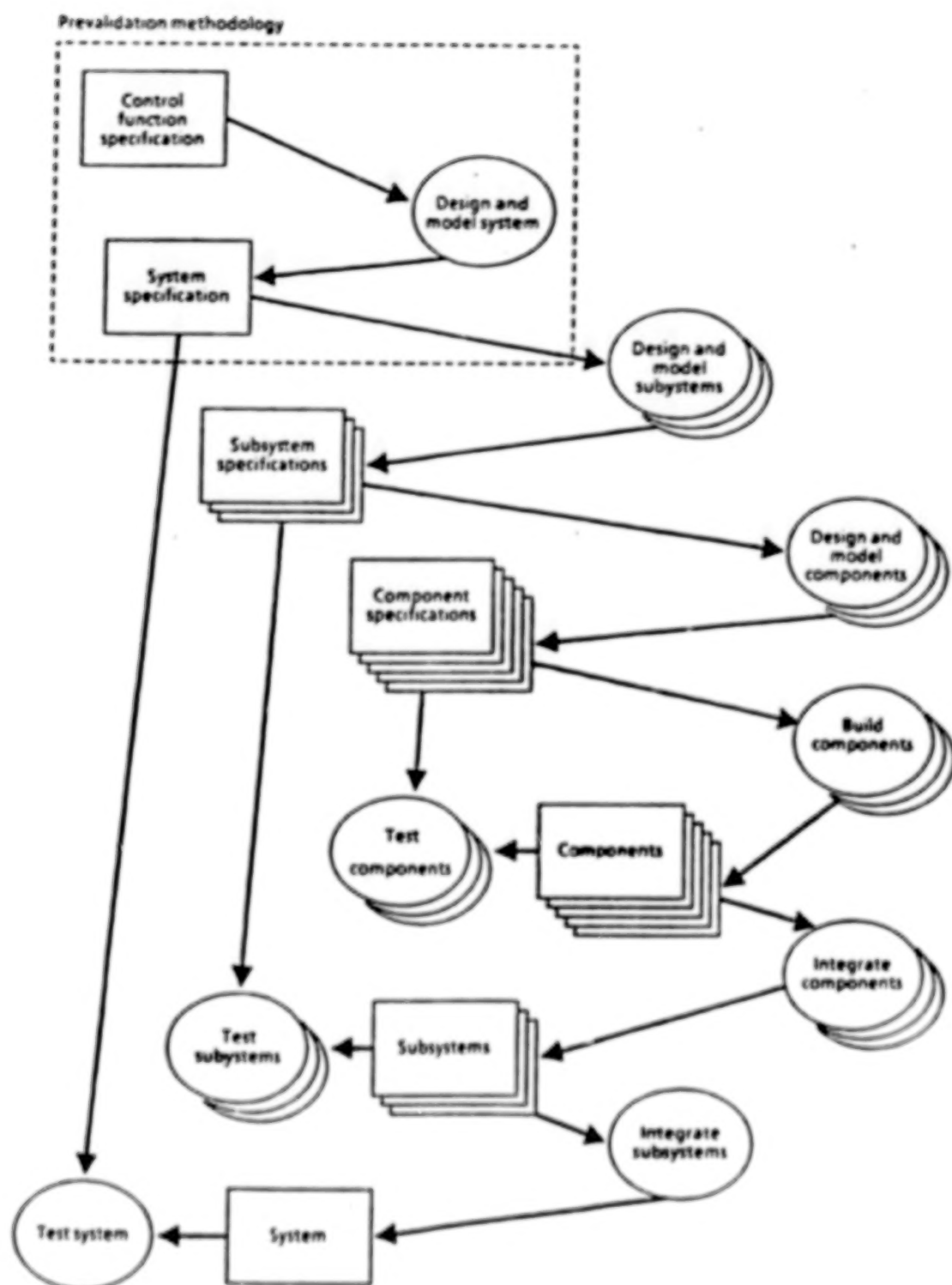
# IAPSA II Prevalidation Methodology



## Prevalidation Methodology

- Early evaluation exposes system weaknesses
- Reliability and performance analysis versus staffing level unresolved
- Methodology allows assessment of cost and technical risk
- Seems to mirror Japanese staffing concept





## **Building-Block Considerations**

### **Contractor/Subcontractor Relationships**

- Requires different approach to subcontractors
- Need to develop:
  - Functional specification
  - Reliability attributes
  - Performance attributes
- Requirements only will not suffice
- Subtleties of building-block interrelationships important

## **Building-Block Considerations Contractor/Subcontractor Relationships (continued)**

- Enforcement of rigor on the vendors
- Do we need a two-step procedure with vendors—
  - During building-block definition
  - During hardware/software bid on system

## Methodology

### Incomplete

- Additional tools
- Maintainability
- Availability
- Survivability
- Cost
- Software
- Tie in to top-level system design
- Relationship between full nonlinear simulation and performance model
- Hardware and software build—subsystem validation and verification
- Lab testing
- Flight testing

## Tools

### Major effort on

- Model development—candidate architecture definition
  - How system works
  - Brief, concise, easy to generate
  - Must include redundancy management operation
- Output data interpretation
  - Complex
  - Very time consuming



## Performance Modeling

- Difficult to simulate
  - Conceptual problem
  - Difficult to implement
    - I/O system service
- Detail of simulation is based upon judgement
- Simulation can validate system architecture
- Verification of model with architecture description
- Simulation used through life cycle
- Unexpected insight via performance simulation

## Reliability Modeling

Methodology goal: rapid evaluation of architecture alternatives

- Current evaluation cycle too slow
- Tools available for ultrareliable systems
  - Short-duration safety
- Long-duration reliability also important
  - Operation with failures
- Common evaluation tool and similar models (safety, mission, etc.) desirable (mandatory?)
- Level of detail and model simplification currently an art
  - Strong pressure toward small and simple models
  - ASSIST/SURE supports techniques for short-duration problems (long-duration?)

Boeing Military Airplanes

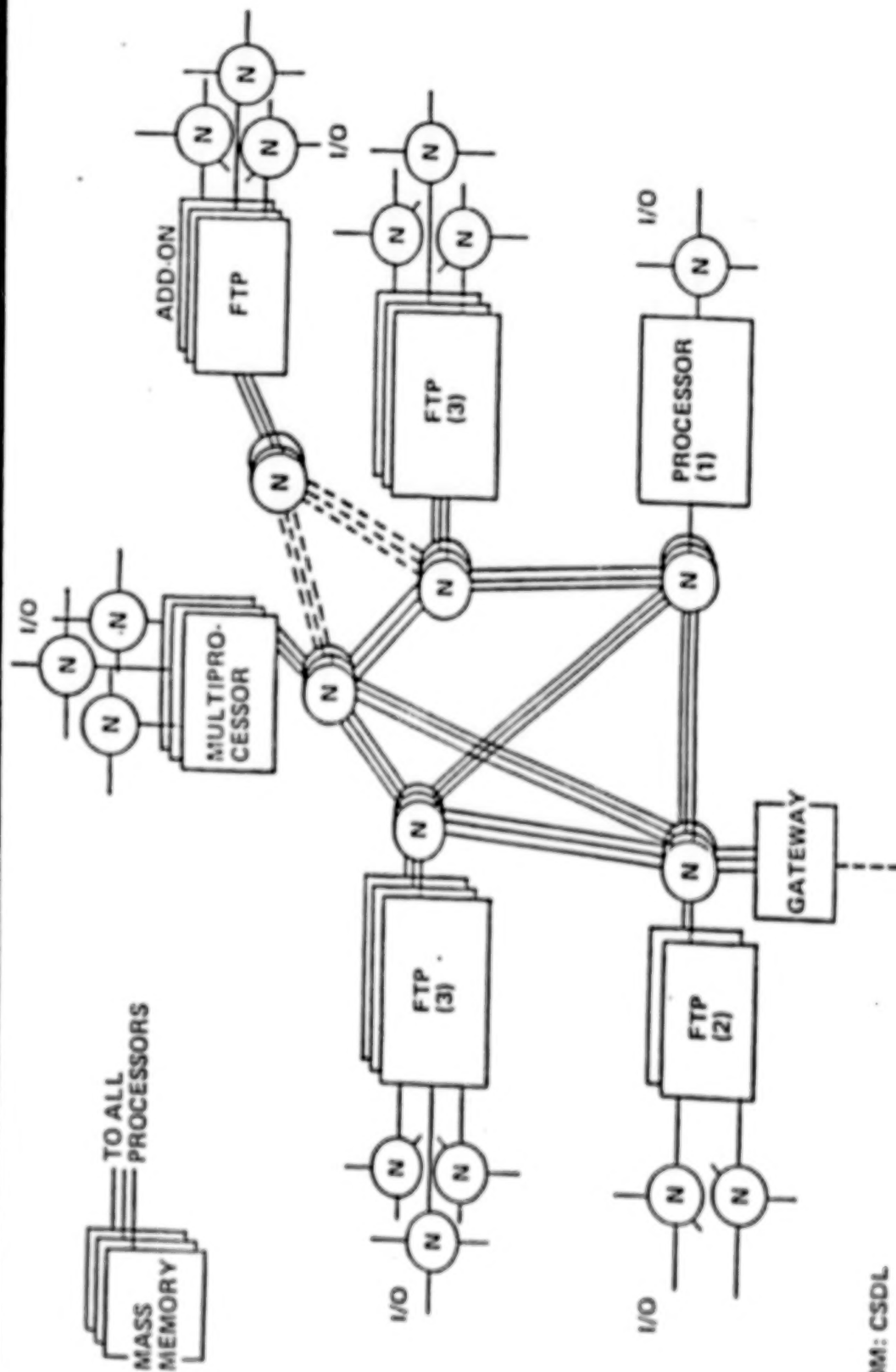
# Advanced Information Processing System

(AIPS)

Designed  
By

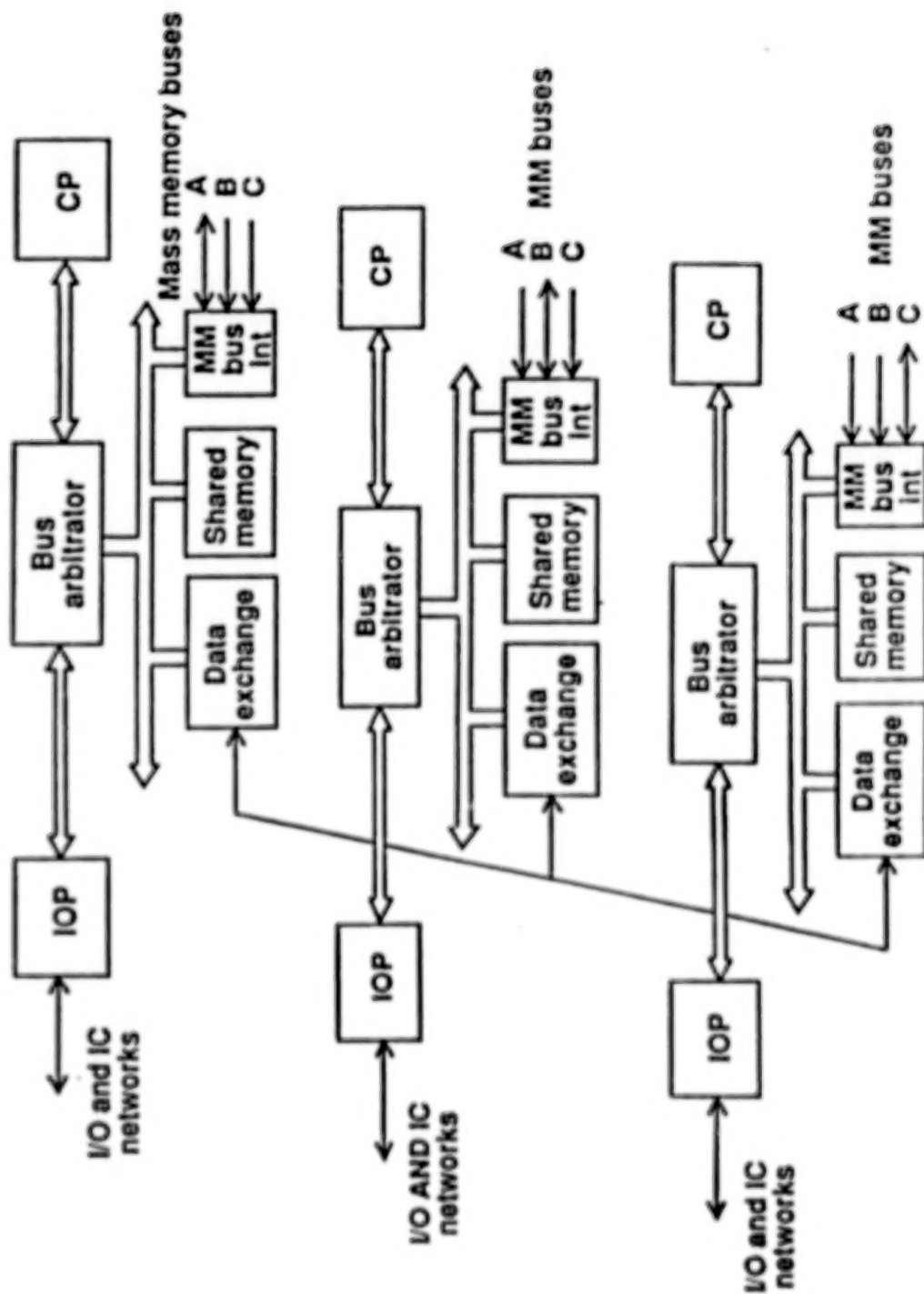
Charles Stark Draper Laboratory

## AIPS Proof-of-Concept Configuration

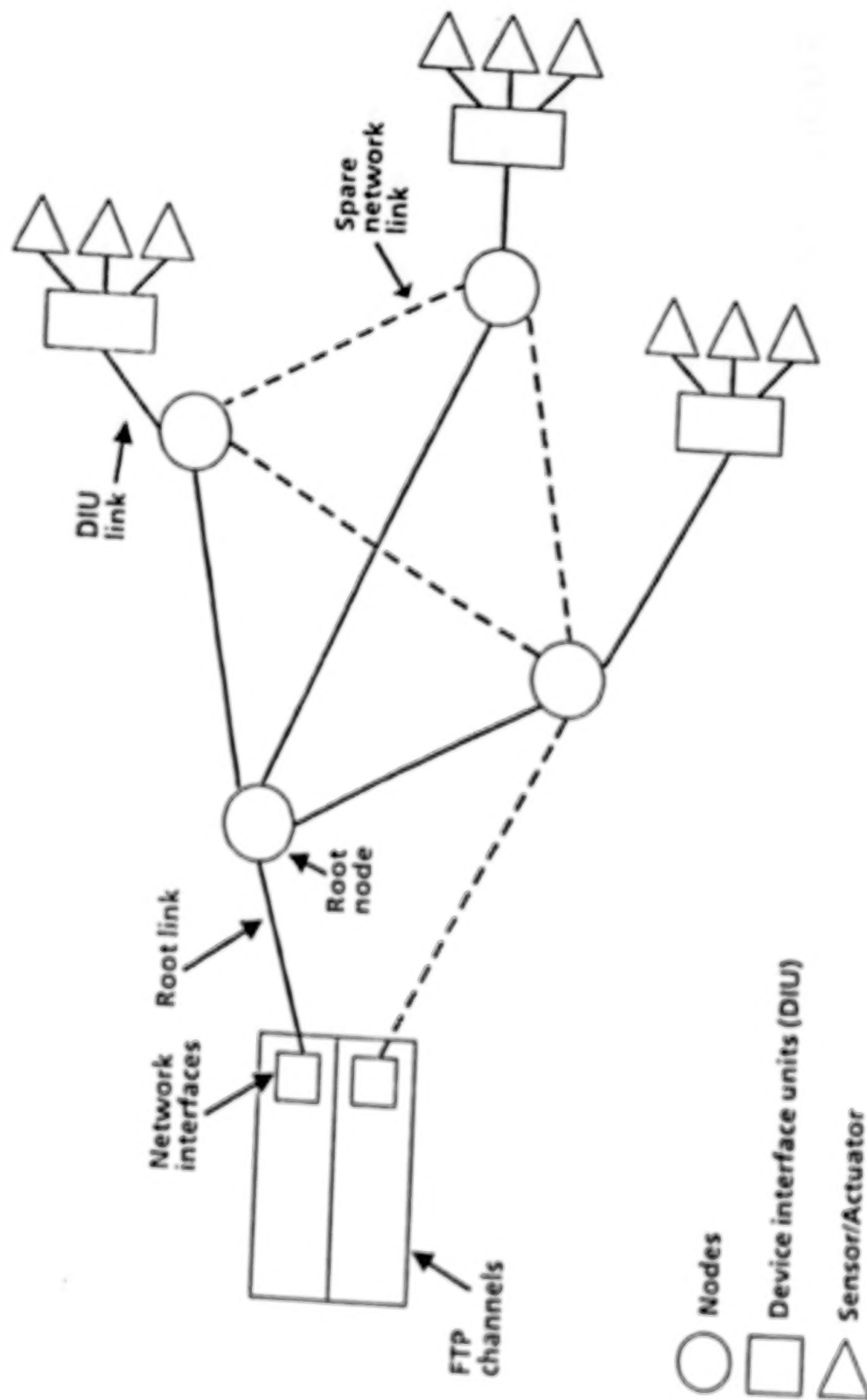


FROM: CSDL

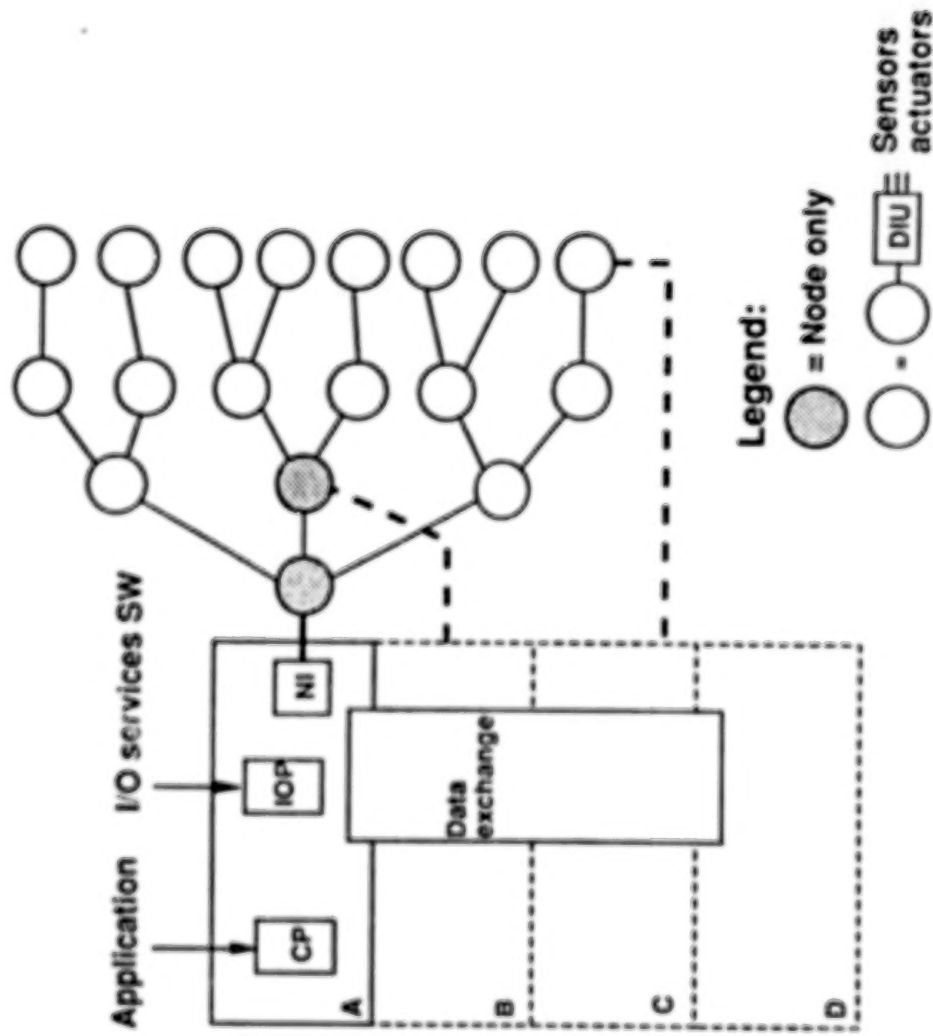
# FTP HW And SW Provide Failure Protection



# I/O Network Elements



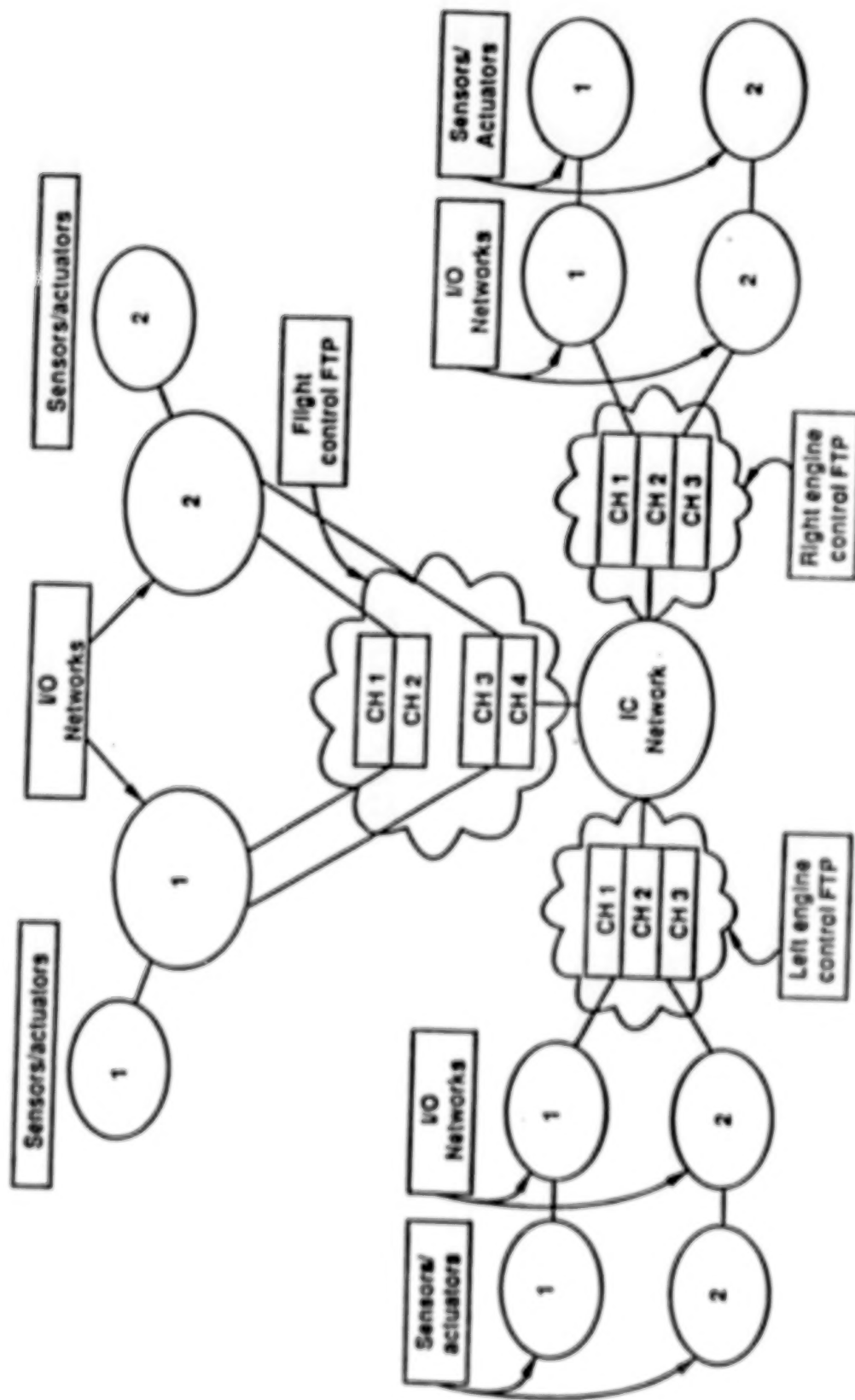
## Flight Control Computer With I/O Connections



# Architecture



# Reference Configuration Overview



## Testing Experience

- Application was quickly integrated into complex fault-tolerant AIPS system
- AIPS simplex application programming model
- CSDL staff assistance
- Impossible to meet goal of testing system with real time performance demands
- Slow time testing focused on system level interactions
- Nonintrusive measurements likely requirement for validation
- During real-time operation with full workload
- System services or operating system functions critical
- Not provided for in original AIPS testing/validation concept

# General Observations Architecture AIPS

## **General Observations Architecture**

- Integrated flight control/propulsion control—feasible
  - Obstacles—mind set problem
- Minimum use of sensors/activators
- Allows for optimum control
- Allows for function migration
- Growth potential
- Subset of Vehicle Management System

## General Observations (Cont)

### AIPS

Very innovative for its time

- Supports true distributed system
- System redundancy transparent to user
- General set of building blocks—user selected
- Fault containment regions

## General Observations (Cont)

### AIPS

- Advantages
  - Building blocks allow expansion with minimum change
  - Building block concept supports common hardware/software throughout the airplane
  - Prevalidated building blocks for both hardware and software
  - Ability to mix elements with different reliability requirements
  - Distributed computing possible
  - Function migration possible
  - Minimizes maintenance and logistic issues

## General Observations (Cont)

### AIPS

- Advantages (cont)
  - System redundancy is inherent in AIPS design
  - Fault containment region is inherent in AIPS design
  - Pre-emptive priority allows application flexibility
  - Communications protocol allows design for minimum sensor/actuator time skew
  - Concept supports dispatch with failures (need faster network repair time)
  - Variation of components within FTP channel (CP/IOP or CP)

## General Observations (Cont)

### AIPS

- Concepts needing attention
  - Insufficient documentation
  - IOP/Data Exchange bottleneck
  - IC network traffic uncertain (not modeled)
  - No discernable difference between network and bus for IAPSA requirements



## General Observations (Cont)

### AIPS

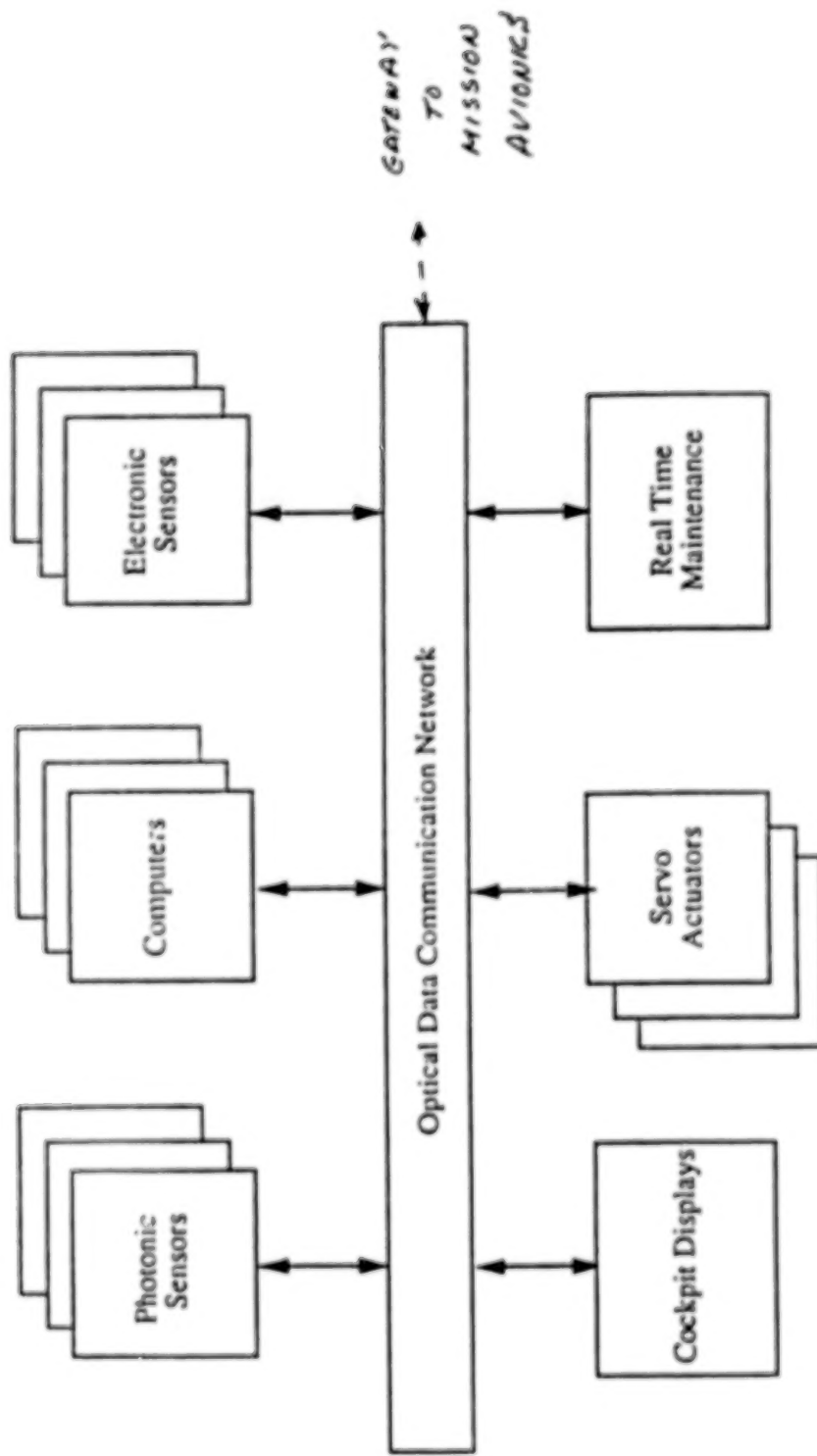
- Concepts needing attention (cont)
- Complex validation issues
  - IO system services
  - IC system services
  - Pre-emptive priority scheduling
- Resynchronization of channel during flight not possible with present design
- System design guidelines not established
- If IC modeled—it appears system would not work with present timing and loading requirements

Boeing Military Airplanes

## Future Systems

## Vehicle Management System

- All flight critical functions
  - Failure causes loss of aircraft
- Near term - military
- Long term - commercial



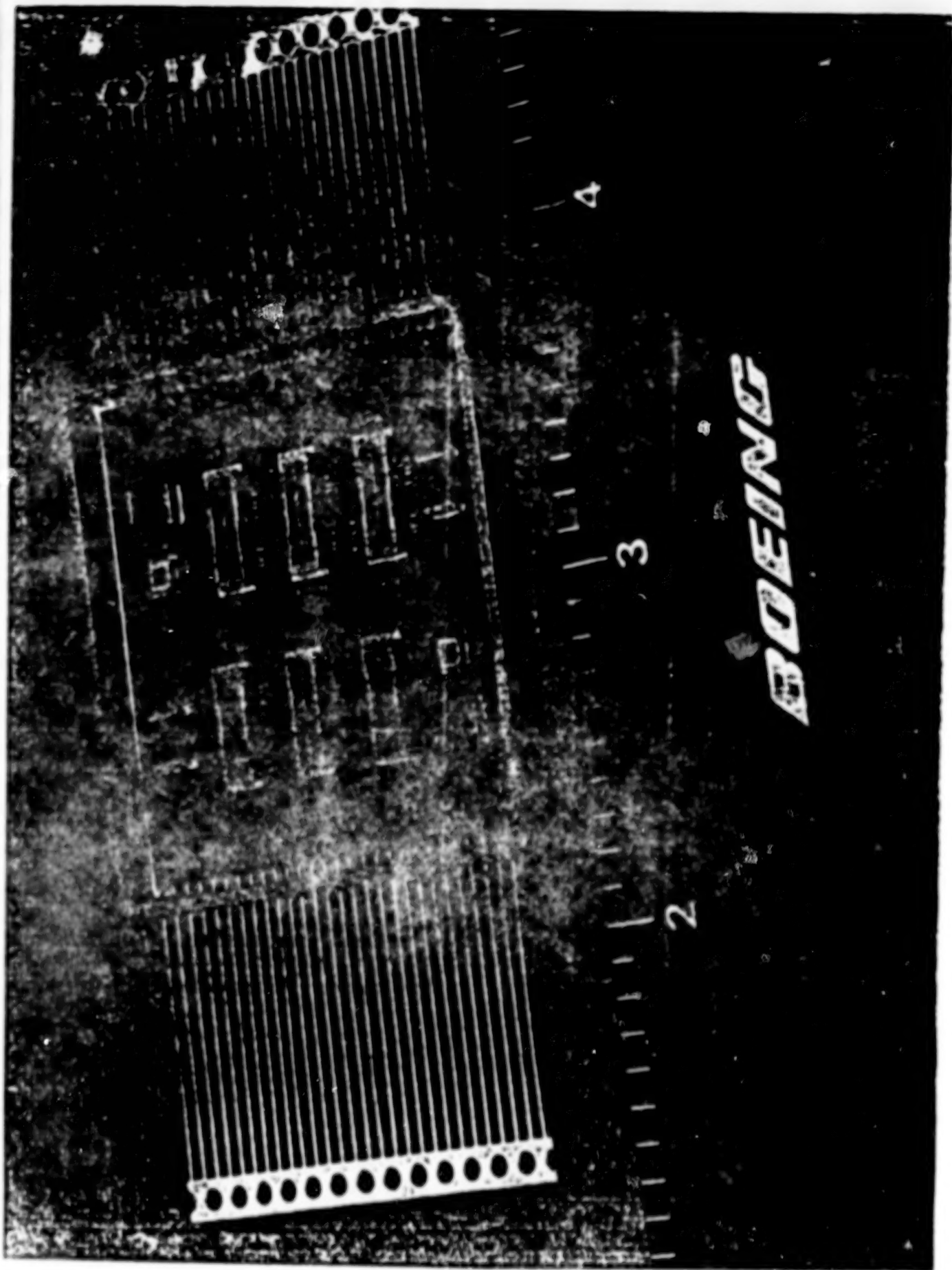
Generic VMS Architecture

## Photonics used for

- Bus
- Sensors
- I/O
- Actuators
- Computers (20 years)

## Benefits of VMS

- Performance
  - Unified environment - coordination of all tasks
- Growth capability
  - Additional nodes - minimum impact
  - Life cycle replacement - minimum topology impact
- Reliability
  - Minimum set of building blocks
  - Minimum part count
  - Common I/O
  - Sharing of sensor data
  - Common redundancy management

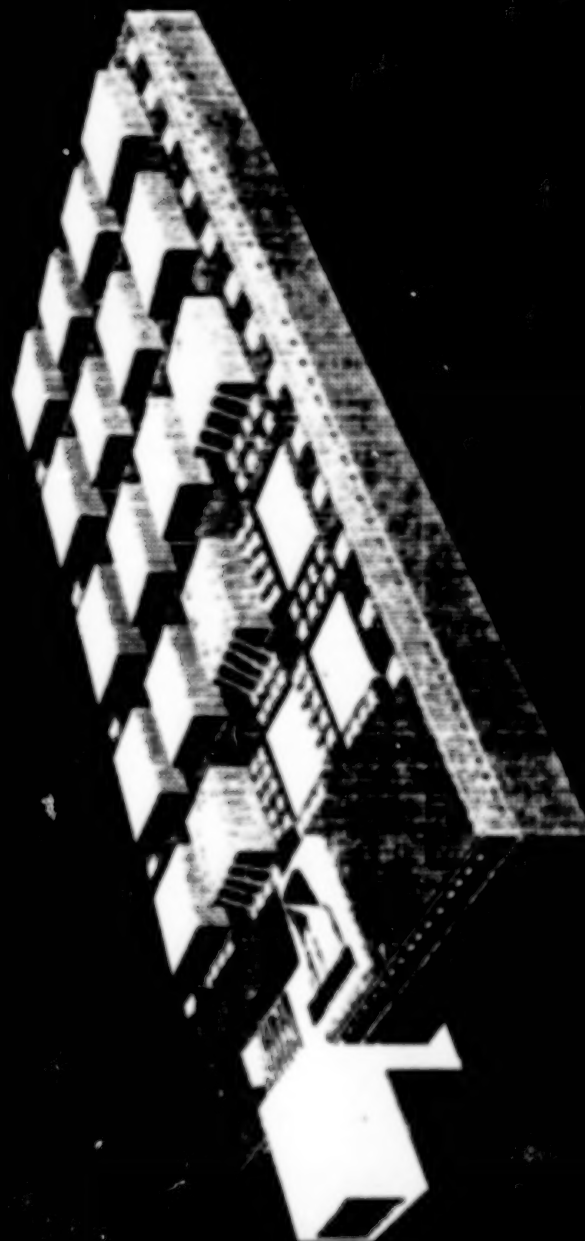


ORIGINAL PAGE IS  
OF POOR QUALITY

# Advanced Multichip Module with Optical Interconnect Technology

ENTER 1912

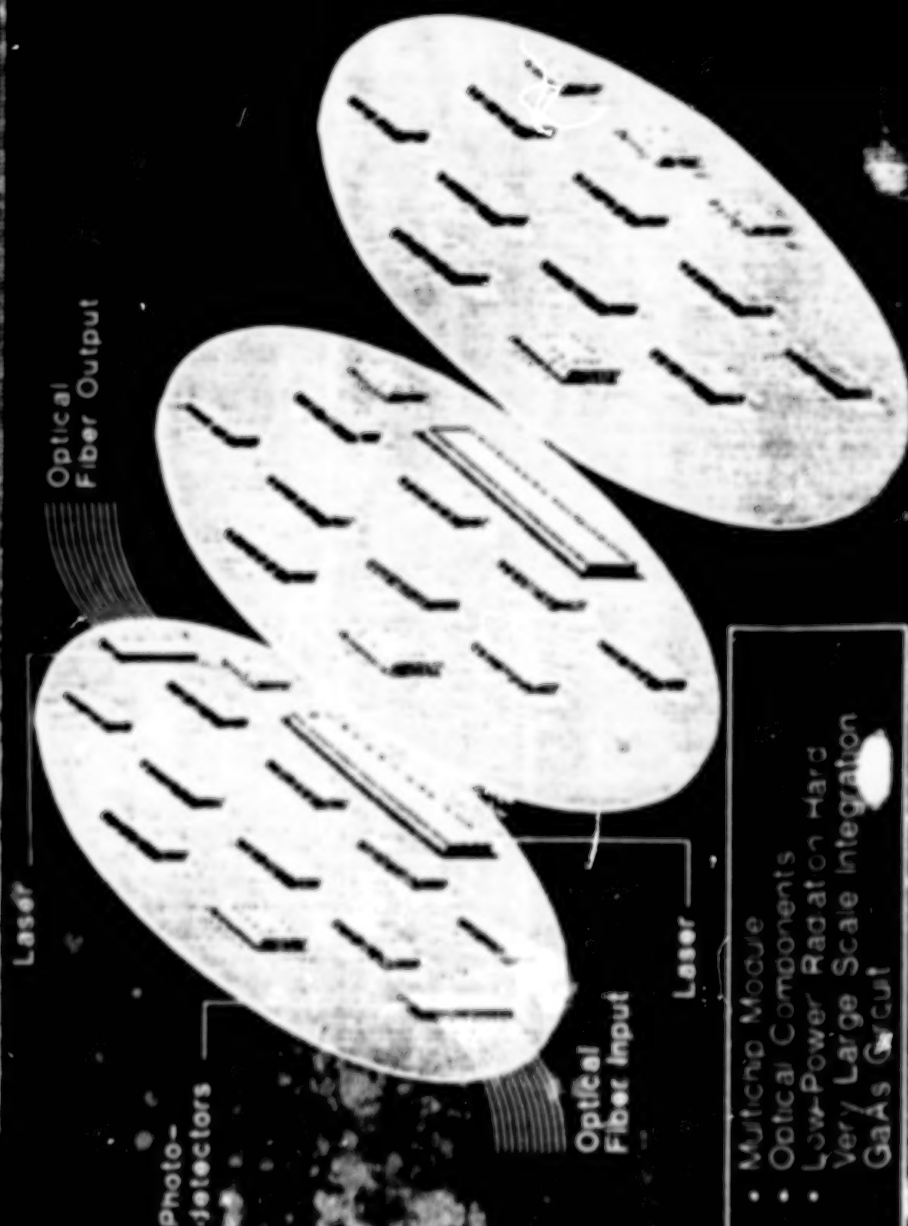
High  
Technology  
Center



ORIGINAL PAGE IS  
OF POOR QUALITY



# A Compact Highly Reliable Avionic Processor



- Multiplex Module
- Optical Components
- Low-Power Radiation Hard
- Very Large Scale Integration
- GaAs Circuit

## **What does all this mean in terms of validation and verification?**

- Design for validation is a critical technology
- Need indepth V&V concurrent with design analysis

## Solution to V&V

Formal Verification - viable solution to the V&V problem for

- Requirements/Specifications
- Hardware
- Software
- System

Boeing Military Airplanes

## Where are we in Formal Verification?

- the following 3 days should tell us!!

94  
7-52  
54-60

319685

N91-17563

**MAFT:**  
**The Multicomputer Architecture for**  
**Fault-Tolerance**

**R. M. KIECKHAFFER**

Computer Science and Engineering  
University of Nebraska - Lincoln  
Lincoln, NE 68588-0115  
(402) 472-2402

[rogerk@fergvax.unl.edu](mailto:rogerk@fergvax.unl.edu)

---

MAFT is a product of the Allied-Signal Aerospace Company, Columbia MD.

## Abstract

This presentation discusses several design decisions made and lessons learned in the design of the Multicomputer Architecture for Fault-Tolerance (MAFT). MAFT is a loosely coupled multiprocessor system designed to achieve an unreliability of less than  $10^{-10}/hr$  in flight-critical real-time applications.

The presentation begins with an overview of the MAFT design objectives and architecture. It then addresses the fault-tolerant implementation of major system functions in MAFT, including Communication, Task Scheduling, Reconfiguration, Clock Synchronization, Data Handling and Voting, and Error Handling and Recovery.

Special attention is given to the need for Byzantine Agreement or Approximate Agreement in various functions. Different methods were selected to achieve agreement in various subsystems. These methods are illustrated by a more detailed description of the Task Scheduling and Error Handling subsystems.

## **Presentation Overview**

---

- **INTRODUCTION**
- **SYSTEM FUNCTIONS**
  - Communication
  - Task Scheduling
  - Task Reconfiguration
  - Clock Synchronization
  - Data Handling and Voting
  - Error Handling and Recovery
- **SUMMARY**

## Design Objectives

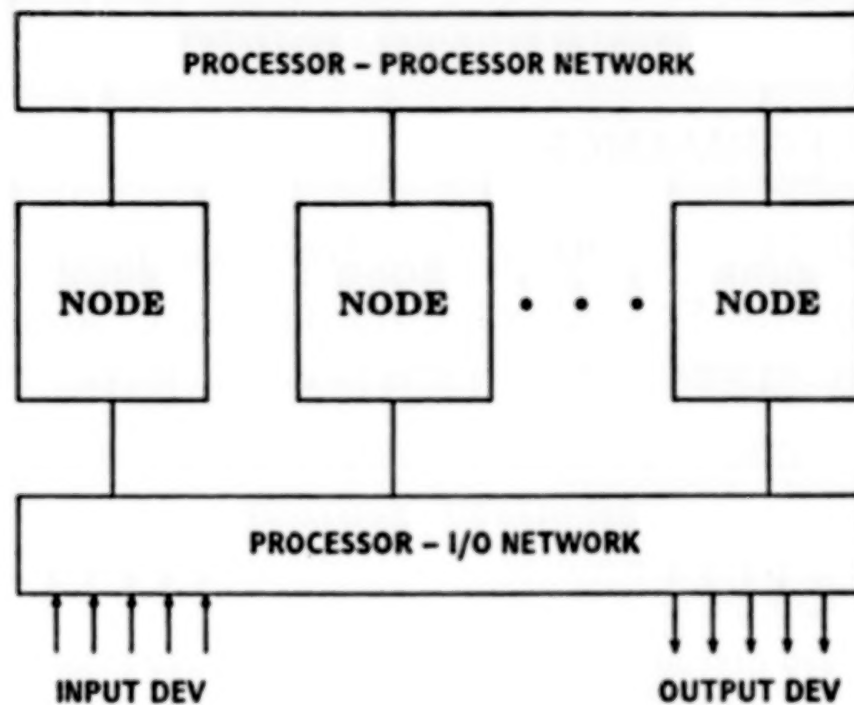
---

- RELIABILITY –  $1.0 \times 10^{-9}$  over 10 hours.
- PERFORMANCE
  - 200 Hz. – Max Task Iteration Rate
  - 5.5 MIPS – Max Computational Capacity
  - 1.0 MBPS – Max I/O Transfer Rate
  - 5.0 ms. – Min Transport Lag (Input → Output)
- REUSABLE
  - Functional Partitioning
    - Application Specific Functions
    - Standard Executive Functions
- LOW EXECUTIVE OVERHEAD
  - Physical Partitioning
    - Separate Executive Processor
    - Hardware Intensive



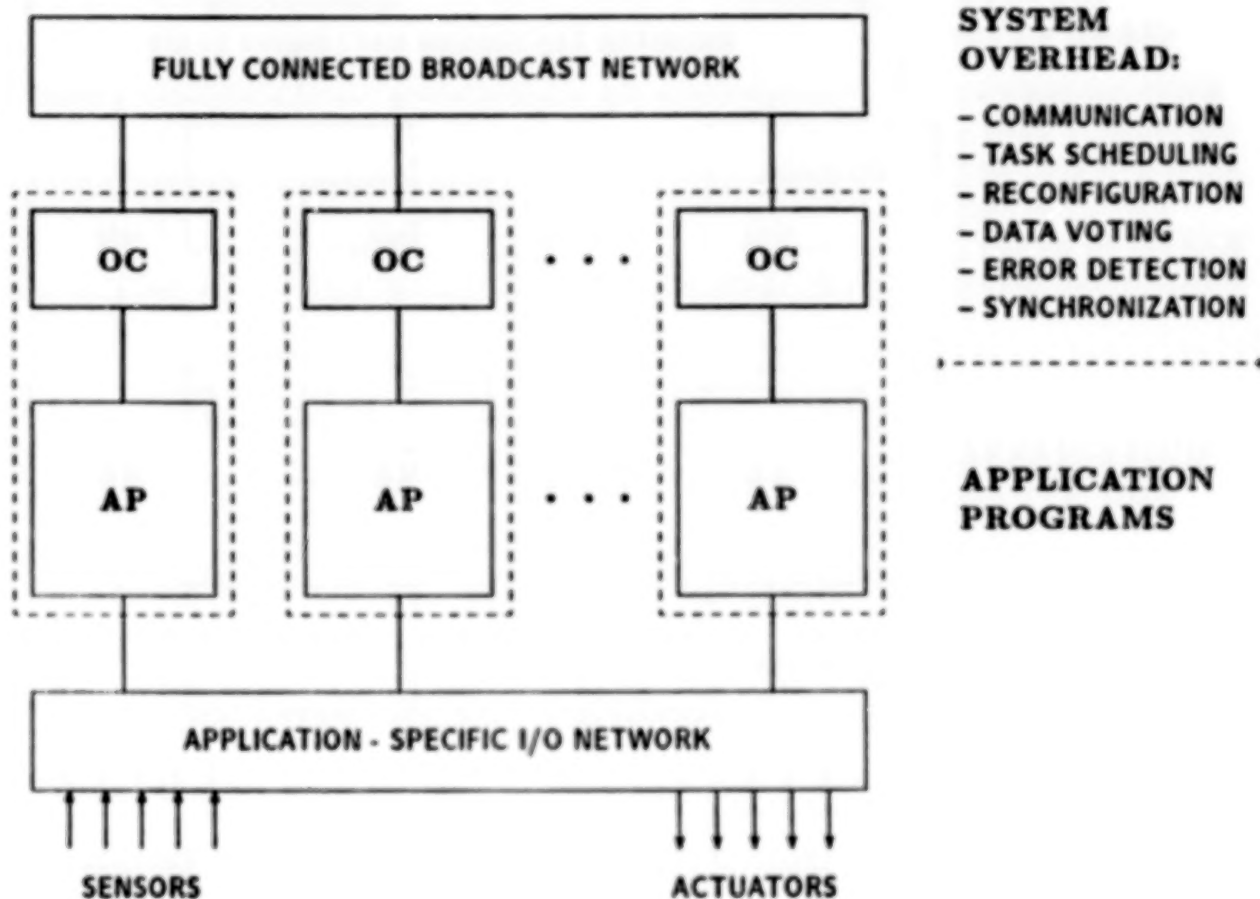
## Loosely-Coupled Multiprocessor

---



- Node  $\Rightarrow$  Processor and Private Memory
- No Shared Memory
- Message-Based Inter-Node Communication
- Common Operating System

# MAFT System Architecture



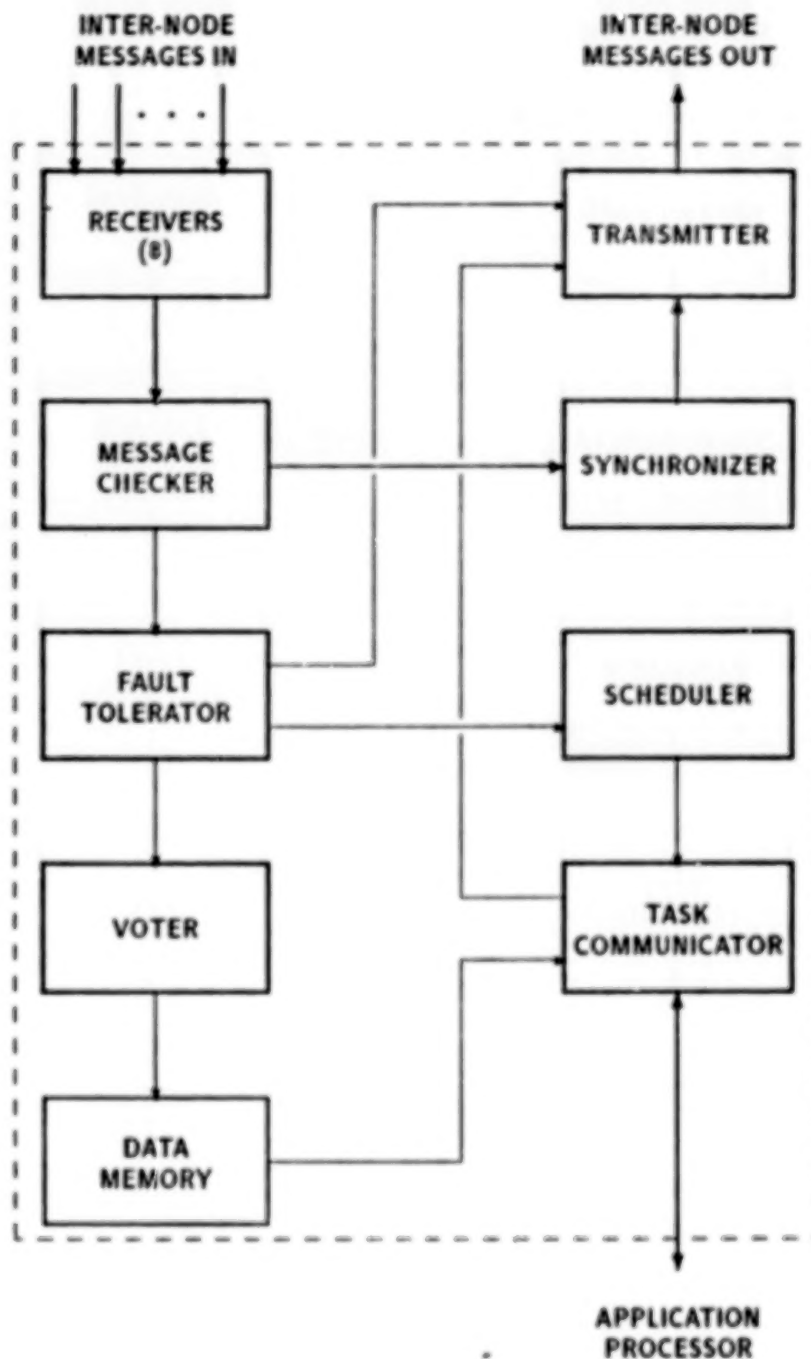
- **OC  $\Rightarrow$  Operations Controller:**

Special Purpose Device Common to All MAFT Systems.

- **AP  $\Rightarrow$  Application Processor:**

General Purpose Application-Specific Processor.

# Operations Controller Block Diagram

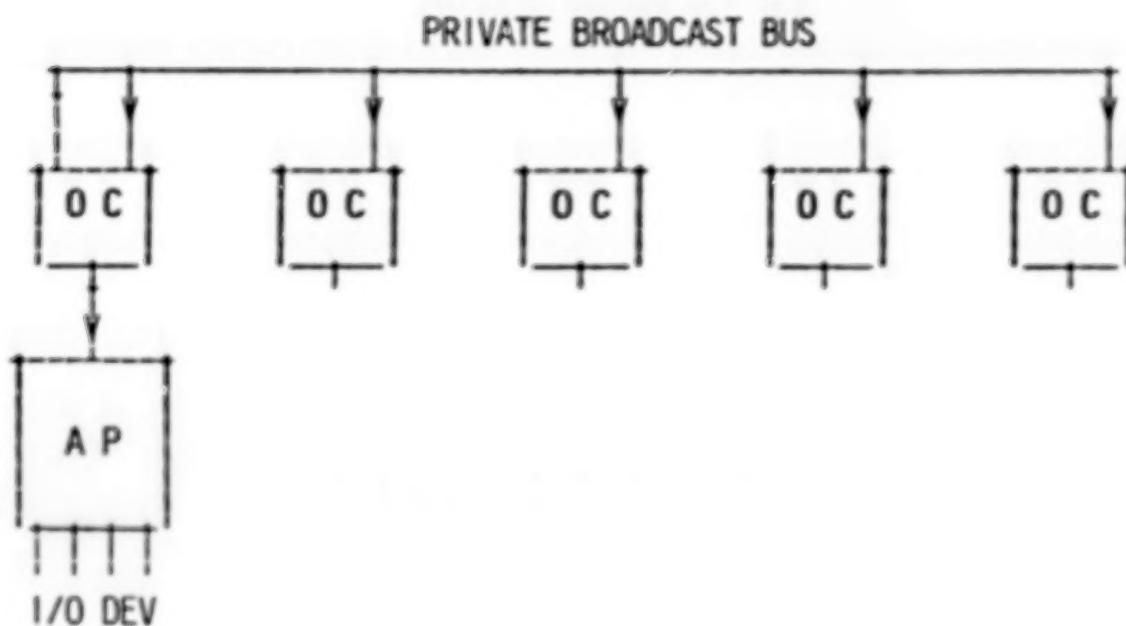


---

# COMMUNICATION

---

## INTER-PROCESSOR COMMUNICATIONS



### - INTRA-NETWORK COMMUNICATION

- MESSAGES TRANSMITTED ON PRIVATE SERIAL BROADCAST BUSSES
- ALL NODES RECEIVE, CHECK AND PROCESS ALL MESSAGES
- MESSAGE TYPES
  - DATA (8/16/32b INT OR BOOL, IEEE STD 32b FLOAT)
  - TASK COMPLETED / STARTED / BRANCH
  - SYNCHRONIZATION / BRANCH INTERACTIVE CONSISTENCY
  - ERROR REPORT

### - OC / AP COMMUNICATION

- 16 BIT ASYNCHRONOUS P.I.O. INTERFACE
- LOOKS LIKE "JUST ANOTHER I/O PORT" TO AP
- COMPATIBLE W/ EXISTING UNIPROCESSOR OPER SYST

## Message Handling

---

- TRANSMITTER

- Format Msg – NID, Msg Type, Framing, ECC
- Broadcast Msg

- RECEIVERS – 1 per incoming link

- Accept Properly Framed Bytes
- Buffer Byte for Message Checker

- MESSAGE CHECKER

- Poll Receivers – 6.4  $\mu s$  cycle
- Physical and Logical Checks
- Steer Good Messages to Other Subsystems
- Dump Bad Messages into "Bit-Bucket"

## LOCAL AP/OC INTERFACE OPERATIONS

### 1. TASK SWITCHING PROCESS

- AP: DONE WITH LAST TASK, WHAT IS THE TASK IDENTIFICATION (TID) NUMBER OF THE NEXT TASK.
- OC: HERE IT IS

### 2. TRANSFER DATA FROM OC TO AP

- AP: GIVE ME THE NEXT INPUT DATA VALUE
- OC: HERE IT IS

### 3. TRANSFER DATA FROM AP TO OC

- AP: HERE'S THE NEXT OUTPUT DATA VALUE
- OC: I GOT IT

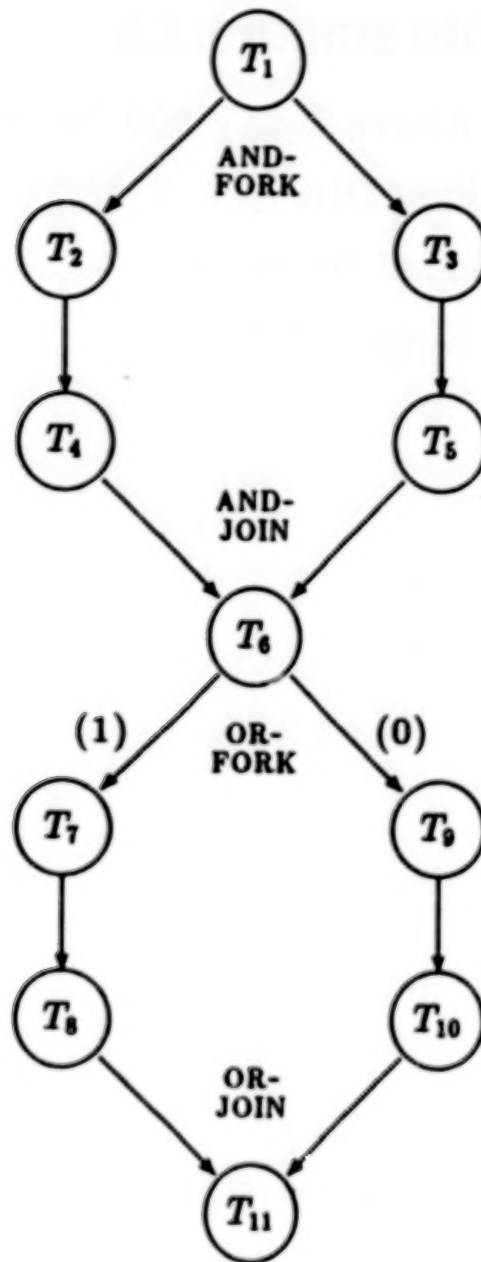
---

## **TASK SCHEDULING**

---



## Typical Task System



## **O.C. View of a Task**

---

- **INTERNAL FUNCTION IS BLACK BOX**
- **VISIBLE PROPERTIES OF A TASK**
  - Priority (static, unique)
  - Iteration Period
  - Precedence Constraints
  - Min and Max duration Limits
  - Fixed Input and Output Shared Data Sets
  - Branch Condition (asserted at completion)

## PERFORMANCE ISSUES

- STRICTLY PERIODIC SCHEDULER

- Fast – Freq Well Above Spec – 500 Hz. vs. 200 Hz.
- Simple – Binary Freq Dist ( $f_i = 2^{-i} f_0$ )
- Flexible – Conditional Branching
- Efficient – Don't Keep AP Waiting

- NON-PREEMPTIVE

- Scheduler Complexity
- Context Switching Time – Unknown Funct of AP
- High Frequencies – Short Tasks

- NO OC INTERRUPTS – I/O

- Scheduler Complexity
- Predictability
- High Frequencies – Polling
- DMA or IOP access to AP Memory

## **FAULT-TOLERANCE ISSUES – I**

- **VARIABLE MODULAR REDUNDANCY**

- Specify Redundancy of Each Individual Task
- Redundancy Matches Criticality
- No More Copies Than Necessary

- **GLOBAL VERIFICATION**

- Consensus Defines Correctness
- All Functions Observable and Predictable
- Replicated Global Scheduler
- Completed/Started (CS) Message:
  - Node I.D.
  - Started Task I.D.
  - Branch Condition

## **FAULT-TOLERANCE ISSUES – II**

### **● DISSIMILARITY BETWEEN COPIES**

- Dissimilar Software and Hardware
  - Guards Against Generic Faults
  - No Guarantee – Knight, Levenson, St. Jean
  - Best Chance of Detecting Error
  - Only Chance of Masking Error
  
- Implications
  - Different Numerical Results
  - Different Execution Times
  
- Impact on Scheduler
  - Min and Max Execution Time Limits
  - Vote on Branch Conditions in CS Messages

## Message Passing Robustness

---

- Delivery NOT GUARANTEED
- Single Msg Error Detect. NOT GUARANTEED
  - ECC coverage  $\geq (1 - 1 \times 10^{-6})$  per msg
- Repeated Undet. Errors PROBABILISTICALLY PRE-CLUDED

## **FAULT-TOLERANCE ISSUES – III**

### **● BYZANTINE AGREEMENT**

- Definition
  - Agreement on All Messages
  - Validity of Agreement
  
- Necessity in MAFT
  - Consensus Defines Correctness
  - Must Have Single Consensus
  
- Preconditions for Disagreement
  - Initial Disagreement – Enhanced by Dissimilarity
  - Assymetric Communication – Minimized by Busses
  
- Solution – Interactive Consistency (Pease et al.)
  - Global Receipt of All Messages
  - Periodic Synchronized Re-Broadcast Rounds
  - Vote on Received Re-Broadcasts
  - Use Voted Values For All Scheduling Decisions

## MAFT Timing Hierarchy

---

PERIOD	SPEC	DEFINITION	BOUNDARY
SUB-ATOMIC	Min 400 $\mu$ s	I.C. Rebroadcast Period  Min Guaranteed Task Duration	Task Inter. Cons. (TIC) Message
ATOMIC	Min 2-2.8 ms	Highest Freq. Task  Clock Sync. Period	System State (SS) Message
GENERAL ITERATION	2 <sup>i</sup> Atom. Per.	Intermed. Freq. Tasks	System State (SS) Message
MASTER	Max 1K Atom. Per.	Lowest Freq. Task	System State (SS) Message



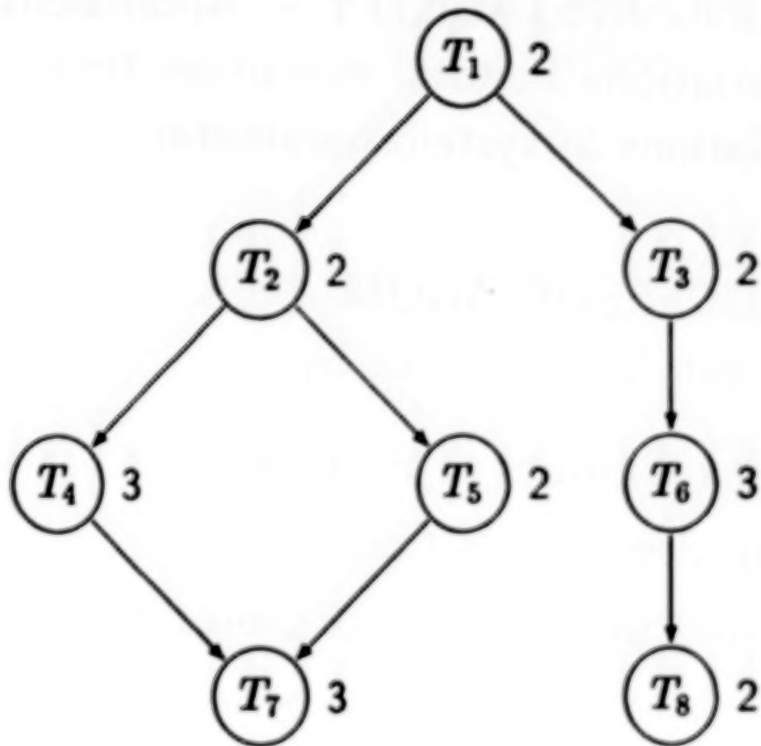
## Scheduling Stability Problem

---

- **SCHEDULING INSTABILITY** – Anomalous or unpredictable variations in total execution time (Makespan) due to variations in system parameters.
- **MULTIPROCESSOR ANOMALIES** – Observation that Makespan can be *increased* by:
  - Increasing Number of Processors,
  - Relaxing Precedence Constraints,
  - Decreasing Individual Task Durations.
- **DYNAMIC FAILURE** – Condition where all tasks execute properly *except* that deadlines are missed.
  - Can occur in a fault-free system,
  - Can be induced by instability.

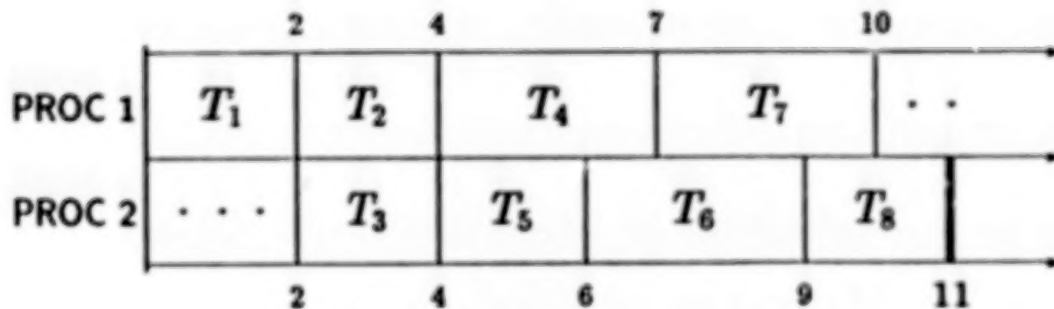
## Sample Task System

---

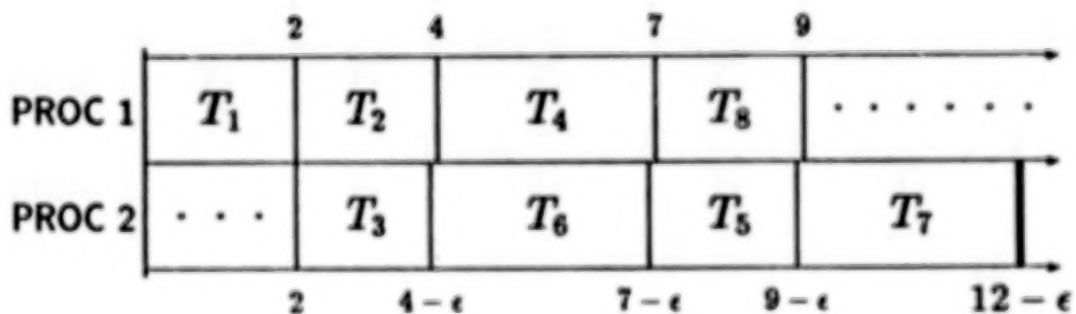


## Instability of Sample Task System

### • STANDARD GANTT CHART (max task durations)



### • NON-STANDARD GANTT CHART (shorten $T_3$ by $\epsilon$ )



### • WHAT HAPPENED?

- $T_3$  finished before  $T_2$ ,
- $T_6$  "ready" before  $T_5$ ,
- $T_5$  displaced by  $T_6 \Rightarrow$  Priority Inversion,
- Critical path ( $T_2 \rightarrow T_7$ ) impeded.

## Previous Work

---

- GRAHAM (1969) – Bound Magnitude of Instability

$$\frac{\omega'}{\omega} = 2 - \frac{1}{N}$$

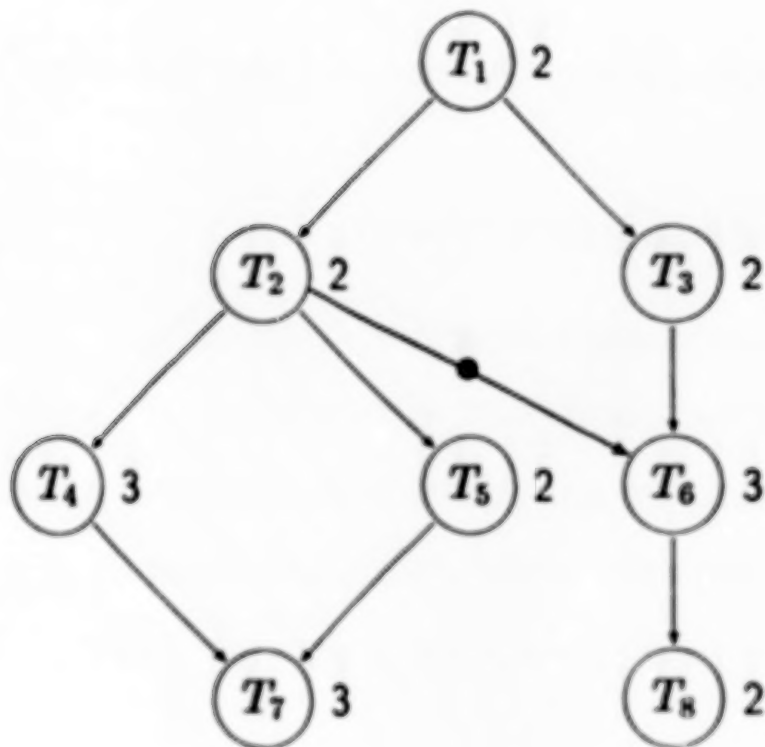
- $\omega$  = Makespan of Standard Gantt Chart,
- $\omega'$  = Makespan of worst-case schedule,
- $N$  = Number of Processors.

- MANACHER (1967) – Stabilization Algorithm

- Necessary Pre-conditions
  - i.  $\exists$  "fork" in Precedence Graph,
  - ii. Successors of forking task run in parallel on Standard Gantt Chart,
  - iii. Possible priority inversion around fork.
- Solution – Impose Artificial Dependency around fork.

## Stabilized Task System

- MANACHER ARTIFICIAL DEPENDENCY ( $T_2 \rightarrow T_6$ )



- EFFECT

- $T_2$  is common parent for both  $T_5$  and  $T_6$ .
- $T_6$  will be "ready" no earlier than  $T_5$ .
- $T_5$  precedes  $T_6$  in priority list,
- $T_6$  can not be selected before  $T_5$ .

## **Limitations of Manacher's Solution**

---

- Sufficient, but not always necessary
- Adds Scheduling Overhead (resolve edge)
- Unrealistic System Model
  - Assumes no scheduler overhead,
  - Assumes dynamic allocation,
  - Allows for no Confirmation Delay,
  - Ignores minimum duration bounds,
  - Does not predict magnitude of instability.

## Current Research

---

- Find Necessary *and Sufficient* Stability Conditions.
- Develop Stabilization Strategies
  - Task System Stabilization
    - Edge Stabilization (Manacher)
    - Vertex Stabilization
    - Hybrid Stabilization
  - Run-Time Scheduler Stabilization
    - Limited Scan Depth
  - Scheduling Algorithm Stabilization
    - Sched. Algorithm Assigns Priorities
    - Constrain to Preclude Necessary Conditions
- Extend System Environment
  - Scheduler Overhead
  - Static Allocation
  - Confirmation Delay
  - Minimum Duration Bounds

---

# SYNCHRONIZATION

---



## MAFT Synchronization

---

- Periodically Exchange System State (SS) Msgs
  - SS Msg  $\Rightarrow$  "Atomic Period" Boundary
  - Synchronization Period = 2 Atomic Periods
- Loosely Synchronized Individual Clocks
  - Msg Exchange  $\Rightarrow$  No Separate Clock Lines
  - Physical Separation  $\Rightarrow$  Damage Tolerance
  - Robustness to "Common Upset" events
- Synchronization Modes
  - Steady State – Maintain Existing Synchronization
  - Warm Start – Converge to Existing Operating Set
  - Cold Start – Form Initial Operating Set
    - Interactive Convergence to synchronize
    - Interactive Consistency  $\Rightarrow$  Steady State
    - Origin of Two-phase algorithm

---

## DATA HANDLING AND VOTING

---

## **Data Management**

---

- DATA GENERATED BY AP
- BROADCAST IN DATA MESSAGE
- RECEIVED AND PROCESSED BY ALL NDOES
  - Static Limit Check
  - On-The-Fly Vote
  - Dynamic Deviance Check

## **On-The-Fly Voting I**

---

- **TRIGGERED BY DATA MESSAGE ARRIVAL**
- **DATA ID ACTS AS UNIQUE VARIABLE NAME**
- **USE ALL PREVIOUS COPIES OF SAME DATA ID**
  - **MS or MME (programmer selectable)**
    - **Sort Serially – High-Order-Bit First**
    - **Select 2 "Medial" Values**
    - **Average (Add and Shift)**
  - **No I.C. Vote for Boolean Types**
    - **Difficult to implement round 2**
    - **Usually Control Data for Mode Switch**
    - **∃ Better Way for Mode Switch**

## **On-The-Fly Voting II**

---

- **DEVIANCE CHECK**

- Compare Each Copy to Voted Value
- Excessive Difference  $\Rightarrow$  error
- Programmer Sets Limits
- Generate Error Vector  $\Rightarrow$  Source Nodes

- **TERMINATE**

- Scheduler Says All Copies Done
- Send Error Vector to Fault-Tolerator
- Send Voted Value to Data Memory
- Swap On-line/Off-line Buffers in Data Memory
- Clear Previously Received Copies from Voter

---

# **ERROR HANDLING AND RECOVERY**

---

## Fault Classifications

---

- BYZANTINE (MALICIOUS)

Pease et al. (1982)

- $N \geq 3t + 1$

- $r \geq t$

- MALICIOUS  $\cup$  BENIGN (self-evident)

Meyer and Pradhan (1987)

- $t = m + b$

- $N \geq 3m + b + 1$

- $r \geq m$

- (ASYMMETRIC  $\cup$  SYMMETRIC)  $\cup$  BENIGN

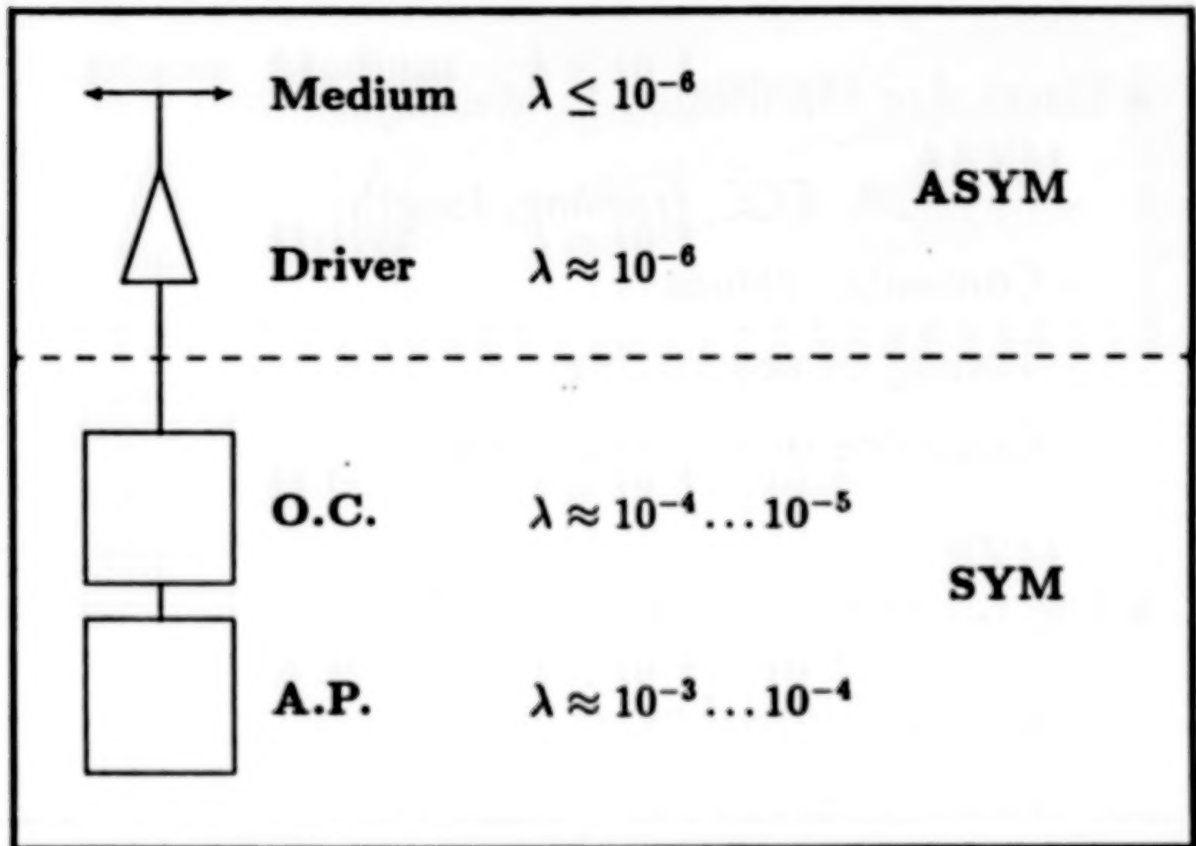
Thambidurai and Park (1989)

- $t = a + s + b$

- $N \geq 3a + 2s + b + r + 1$

- $r \geq a$

## Fault Classes by Source



- Can Estimate Separate  $\lambda$ 's

- $\lambda_{asym} \approx 10^{-6}$
- $\lambda_{sym} \approx 10^{-3} \dots 10^{-4}$

- Generic Fault = Multiple Symmetric

- $\lambda_{gen} \approx 10^{-5} ?$



## Error Detection

---

- Errors Are Manifested In Messages
  - Physical: ECC, framing, length
  - Contents: values
  - Timing or sequencing
  - Existence or non-existence
- Log Errors Over One Atomic Period
  - Errors reported by all subsystems
  - Fault-Tolerator records errors
  - $\exists$  31 separate error "flags"
  - $\exists$  Unique "Penalty Weight"  $PW$  for each flag
  - $\exists$  "Incremental Penalty Count"  $IPC$  for each node
  - FOR each flag  $f$  reported against node  $i$ :
    - $IPC(i) := IPC(i) + PW(f)$

## Error Reporting

---

- Broadcast  $ERR(i)$  Message
  - At beginning of next Atomic Period
  - Contents:
    - $IPC(i)$
    - $BPC(i)$  – Base (current) penalty count
    - All Error Flags for node  $i$
  
- No ERR Message  $\Rightarrow$  No Detections

## BPC Manipulation

---

- BPC  $\Rightarrow$  Health Of Node
- Increasing BPC – ERR Message Vote
  - Vote on  $BPC(i)$
  - Vote on  $IPC(i)$
  - $BPC(i) := BPC(i) + IPC(i)$
- Decreasing BPC – Fixed decrement
  - $\exists$  Penalty Decrement value  $PD$
  - At New Master Period
  - $BPC(*) := BPC(*) - PD$
  - Allows For Eventual Readmission

## Exclusion/Readmission

---

- Recommend Exclusion/Readmission

- $\exists$  Exclusion Threshold  $T_{excl}$
- $\exists$  Admission Threshold  $T_{adm}$
- Recommend in next SS message:
  - $BPC(i) \geq T_{excl} \Rightarrow$  Exclude  $i$
  - $BPC(i) \leq T_{adm} \Rightarrow$  Readmit  $i$
  - $T_{adm} < BPC(i) < T_{excl} \Rightarrow$  No Change

- I.C. Vote on Recommendations

- Consistent System State is Critical
- Free (needed for cold-start)
- Highly Degraded Systems
- Common Mode Upset Recovery



## **Sed Quis Custodiet ...III**

---

- **AP – Diagnostics in Workload**
- **OC – System Level Self-Test**
  - **Errors Very Rare**
  - **Inject Faults to Exercise Error Detection**
    - **Special self-test Task ID**
    - **Suspend normal Transmitter Ops**
    - **Transmit string from self-test ROM**
    - **Can transmit ANY test scenario**
  - **Test Results Based On**
    - **False/Missed Accusations**
    - **Cyclic Link Check**
  - **Independent of Actual Bit-Stream**
  - **Rotate "Originator" Duty**
  - **Complete Coverage If ANY One Node Correct**

## Version Management

---

- SSV = System State Vec – eg (2,1,1)
- VMV = Version Management Vec – eg (1,1,1)
- WMV = Workload Management Vec – (SSV) or (VMV)
- Vectors Used By Different Subsystems

---

Data Voter	VMV	Inactive Copy Ignored For Vote
Dev Checker	SSV	Inactive Copy Still Monitored
Scheduler	WMV	Inactive Copy May Not Run

---

- WMV = SSV
  - Inactive Copy Still Executing
  - Actual Tasks Being Monitored
  - Best for Generic Fault Detection
- WMV = VMV
  - Inactive Copy Doing Something Else
  - Will Not Be Affected By Generic
  - Can Activate To Replace Sibling
  - Best For Generic Recovery

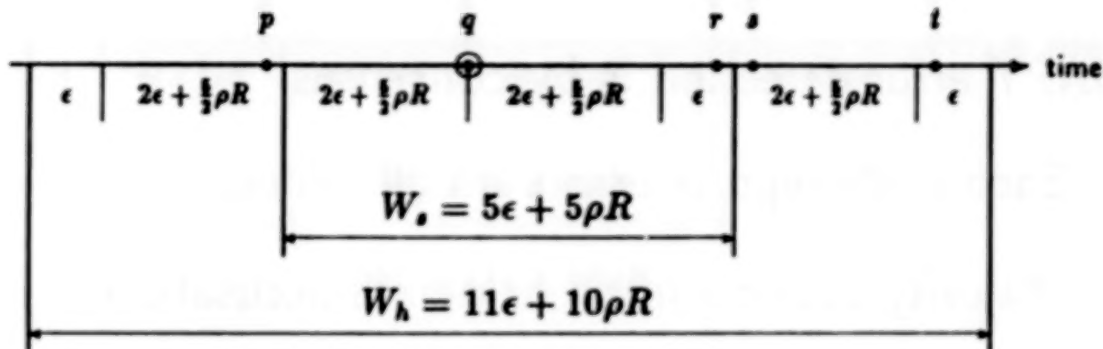
## **Synchronizer Error Detection**

---

- **MAFT error detection is by consensus**
  - Each node reports errors on all nodes.
  - Majority vote confirms or denies accusations.
  - Disagreement with majority may itself be an error.
  
- **Faulty node must be detected by majority of nodes**
  - Must be "far enough" out of sync
  - There exists a region of ambiguity
  - Defines size of "Sync Window"



## Synchronizer Error Windows



- $W_s$  = SOFT ERROR WINDOW

- Spans Range of Receipts from Non-Faulty Nodes
- Error May Not Be Confirmed
- Inherent Ambiguity
- Must Suspend Error Disagreement Penalties

- $W_h$  = HARD ERROR WINDOW

- IF Any non-faulty node detects a Hard-Error  
THEN All non-faulty nodes detect an Error
- Can demand Corroboration

## Typical Sync. Window Values

---

- $\epsilon = 7 \mu\text{sec} - 600 \text{ ft. separation}$
- $\rho = 5 \cdot 10^{-5}$
- $R = 20 \text{ msec} \Rightarrow 10 \text{ msec Atomic Pd.} \Rightarrow 100 \text{ Hz.}$
- $\rho R = 1 \mu\text{sec}$
- No Faults:  $\text{Max } \delta = 8.5 \mu \text{ sec}$
- With Faults:  $\text{Max } \delta = 16.5 \mu \text{ sec}$
- $W_s = 40 \mu \text{ sec}$
- $W_h = 87 \mu \text{ sec}$

---

## SUMMARY

---

## SUMMARY COMMENTS ON THE APPLICATION OF MAFT TECHNOLOGY

### 1. CAPABILITIES

- BASIS OF A GENERIC REAL-TIME MULTICOMPUTER SYSTEM
- REMOVES F.T. OVERHEAD FROM APPLICATION PROCESSOR
- HANDLES ALL REDUNDANCY MANAGEMENT WITHIN COMPUTER
- ASSISTS IN REDUNDANCY MANAGEMENT OF I/O SYSTEM

### 2. FLEXIBILITY

- INDEPENDENT OF I/O ARCHITECTURE
- HIGHLY RECONFIGURABLE AND GRACEFULLY DEGRADABLE
- PROVIDES MECHANISMS, NOT POLICIES

### 3. USABILITY

### **ADVANTAGES OF APPROACH**

- PARTITIONED APPROACH SIGNIFICANTLY REDUCES PROCESSOR OVERHEAD
- DATA DRIVEN ARCHITECTURE MUCH FASTER THAN SOFTWARE IMPLEMENTATION
- NOT DEPENDENT UPON ARCHITECTURE OF APPLICATION PROCESSOR
- REDUNDANCY IS "TASK-BASED" AND FLEXIBLE
- SUITABLE FOR HIGH RELIABILITY AND HIGH PERFORMANCE APPLICATIONS

APRIL 1, 1985

# MAFT Bibliography

---

## References

- [Dar88] Darwiche, A.A., and F.M. Doerenberg, "Application of the Bendix/King Multicomputer Architecture for Fault-Tolerance in a Digital Fly-By-Wire Control System", *Midcon*, Aug 1988.
- [Glu86] Gluch, D.P., and M.J. Paul, "Fault-Tolerance in Distributed Digital Fly-by-Wire Flight Control Systems", *AIAA/IEEE Seventh Digital Avionics Systems Conference*, Oct 1986.
- [Kie87] Kieckhafer, R.M., "Task Reconfiguration in a Distributed Real-Time System," *Eighth IEEE Real-Time Systems Symposium*, Dec 1987.
- [Kie88] Kieckhafer, R.M., et al, "The MAFT Architecture for Distributed Fault-Tolerance", *IEEE Trans. Computers*, V. C-37, No. 4, pp. 398-405, Apr 1988.
- [Kie89] Kieckhafer, R.M., "Fault-Tolerant Real-Time Task Scheduling in the MAFT Distributed System," *Proc. Twenty-Second Hawaii International Conference on System Sciences*, Jan 1989.
- [McE88] McElvany, M.C., "Guaranteeing Deadlines in MAFT," *Proc. IEEE Real-Time Systems Symp.*, pp. 130-139, Dec 1988.
- [Tha88] Thambidurai, P.M., and Y.K. Park, "Interactive Consistency with Multiple Failure Modes", *Proc. Seventh Reliable Dist Systems Symp.*, Oct 1988.
- [Tha88a] Thambidurai, P.M. *Critical Issues in the Design of Distributed, Fault-Tolerant, Hard Real-Time Systems*, Ph.D. Dissertation, Dept. of Electrical Engineering, Duke University, 1988.
- [Tha89] Thambidurai, P.M., et al., "Clock Synchronization in MAFT", *Nineteenth Fault-Tolerant Computing Symposium*, pp. 142-151, Jun 1989.
- [Wal88] C.J. Walter, "MAFT: An Architecture for Reliable Fly-by-Wire Flight Control," *Proc. AIAA/IEEE Eighth Digital Avionics Systems Conference*, pp. 415-421, Oct 1988.

146  
55-61 P19  
219686

**Design For Validation**  
Based on Formal Methods

Ricky W. Butler

NASA Langley Research Center  
Hampton, VA 23665

N91-17564

## VALIDATION OF ULTRA-RELIABLE SYSTEMS

### DECOMPOSES INTO TWO SUBPROBLEMS

1. Quantification of probability of system failure due to physical failure
2. Establishing that **DESIGN ERRORS** are not present <sup>1</sup>.

---

<sup>1</sup>(note. Quantification of 2 is infeasible)



## Achieving Ultra-Reliable Software (Approaches)

- Testing (Lots of it)
- Design Diversity (e.g. N-version programming)
- Fault Avoidance:
  - Formal Specification/Verification
  - Automatic Program Synthesis
  - Reusable Modules

## Life-Testing

### Basic Observation:

$10^{-9}$  Probability of failure estimate for a 1 hour mission

### REQUIRES

$> 10^9$  hours of testing

(  $10^9$  hours = 114,000 years )

## Design Diversity

1. Separate Design/Implementation Teams
2. Same Specification
3. Multiple Versions
4. Non-exact Threshold Voters
5. Hope design flaws manifest errors independently or nearly so.

## The Big Problem For Design Diversity Advocates

- experiments in the low-nominal reliability region have shown that the number of near-coincident failures far exceeds the number predicted by an independence model.
- Certainly independence cannot be assumed axiomatically for ultrareliable regime
- If cannot assume independence must measure correlations.

Quantification of N-version programs not feasible in the ultrareliable regime

- Since one cannot assume independence, it must be treated as black box
- Back to life-testing problem again
- Any alternative model would have to be validated. But How?

How do we get ultra-reliable numbers for hardware (physical failure)?

THE ONLY THING THAT ENABLES QUANTIFICATION OF ULTRA-RELIABILITY FOR H/W IS THE

**INDEPENDENCE ASSUMPTION !!**

**- THE INDEPENDENCE ASSUMPTION CANNOT BE DEMONSTRATED FOR MULTI-VERSION S/W IN THE ULTRA-RELIABLE REGION**

## The Danger of Design Diversity (N-version Programming, Recovery Blocks, etc.)

- creates an "illusion" of ultra-reliability. By assuming independence, the advocates of S/W fault-tolerance generate ultra-high estimates of reliability.
- As long as industry/certification agencies believe that S/W fault-tolerance will solve the problem, formal methods will not be pursued.

## Design For Validation

1. Designing a system in a manner that a complete and accurate reliability model can be constructed. All parameters of the model which cannot be deduced from the logical design must be measured. All such parameters must be measurable within a feasible amount of time.
2. The design process makes tradeoffs in favor of designs which minimize the number of measurable parameters in order to reduce the validation cost. A design which has exceptional performance properties yet requires the measurement of hundreds of parameters (e.g say by time-consuming fault-injection experiments) would be rejected over a less capable system that requires minimal experimentation.
3. The system is designed in a manner that enables a proof of correctness of its logical structure. Thus, the reliability model does not include transitions representing design errors.
4. The reliability model is shown to be accurate with respect to the system implementation. This is accomplished analytically.



## Illustration 1

Suppose we must design a simple fault-tolerant system with a probability of failure no greater than  $2 \times 10^{-6}$  whose maximum mission time is 10 hours.

- We quickly eliminate the use of a simplex processor since there is no technology which can produce a processor with this low of a failure rate.
- Thus, we begin to explore the notion of fault-tolerance. We next consider the use of redundancy—how about a dual? When the first processor fails, we will automatically switch to the other processor.



Unfortunately, our design suffers from one major problem. It is **impossible** to *prove* that any implementation behaves in accordance with this model.

The problem is that one cannot design a dual system which can detect the failure of the first processor and switch to the second 100% of the time. Thus, we must accept the fact that there is a single-point failure in our system and include it in our reliability model

## SURE Run

Now we have a parameter in our model which must be measured— $C$ . It represents the fraction of single faults from which the system successfully recovers. Can this parameter can be measured in a feasible amount of time (i.e. say less than year) with statistical significance?

\$ sure

SURE V7.5 NASA Langley Research Center

1: read dualspt

2:

3: LAMBDA = 1E-4;

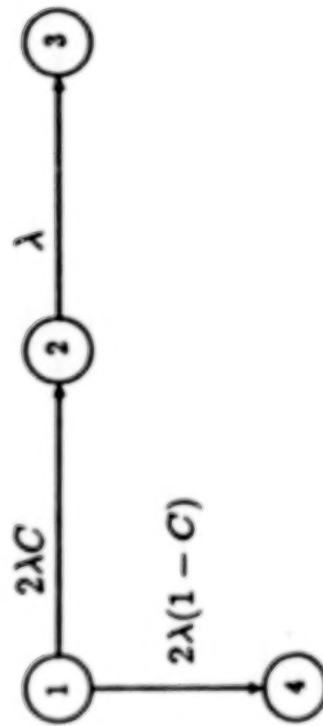
4: C = .9 TO 1 BY 0.01;

5: 1.2 = 2\*LAMBDA\*C;

6: 2.3 = LAMBDA;

7: 1.4 = 2\*(1-C)\*LAMBDA;

8: run



C	LOWERBOUND	UPPERBOUND
9.00000e-01	2.00699e-04	2.00900e-04
9.10000e-01	1.80729e-04	1.80910e-04
9.20000e-01	1.60759e-04	1.60920e-04
9.30000e-01	1.40789e-04	1.40930e-04
9.40000e-01	1.20819e-04	1.20940e-04
9.50000e-01	1.00849e-04	1.00950e-04
9.60000e-01	8.08790e-05	8.09600e-05
9.70000e-01	6.09090e-05	6.09700e-05
9.80000e-01	4.09390e-05	4.09800e-05
9.90000e-01	2.09690e-05	2.09900e-05
1.00000e+00	9.99000e-07	1.00000e-06

From this run we know that  $C$  must be between 9.9 and 1.0 in order to meet our reliability goal. We rerun the model to get a closer value:

## 2nd Run

\$ sure

9? read dualspz

10: LAMEDA = 1E-4;  
11: C = .999 TO 1 BY 0.0001;  
12: 1.2 = 2\*LAMEDA\*C;  
13: 2.3 = LAMEDA;  
14: 1.4 = 2\*(1-C)\*LAMEDA;

0.02 SECS. TO READ MODEL FILE

16? run

C	LOWERBOUND	UPPERBOUND	COMMENTS	RUN #2
9.99000e-01	2.99600e-06	2.99900e-06		
9.99100e-01	2.79630e-06	2.79910e-06		
9.99200e-01	2.59660e-06	2.59920e-06		
9.99300e-01	2.39690e-06	2.39930e-06		
9.99400e-01	2.19720e-06	2.19940e-06		
9.99500e-01	1.99750e-06	1.99950e-06		
9.99600e-01	1.79780e-06	1.79960e-06		
9.99700e-01	1.59810e-06	1.59970e-06		
9.99800e-01	1.39840e-06	1.39980e-06		
9.99900e-01	1.19870e-06	1.19990e-06		
1.00000e-00	9.99000e-07	1.00000e-06		

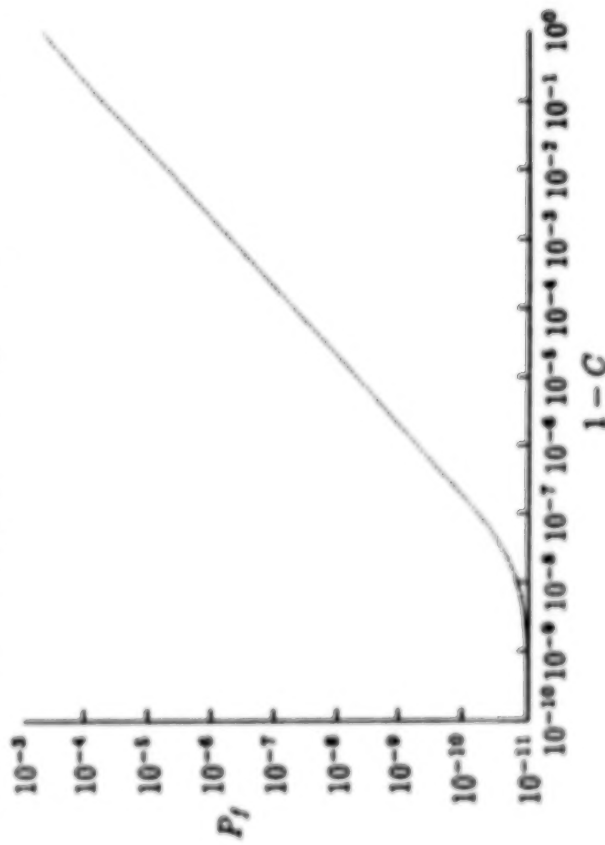
- Now, we can see that we must demonstrate that C is greater than 0.9995.
- it can shown that 20000 observations are necessary to estimate this parameter to a reasonable level of statistical significance.
- If we optimistically assume that each fault injection requires 1 minute, then this validation exercise would require 330 hours (i.e. 14 days).

In this case, we decide we can live with this amount of testing and proceed to develop our system.

## Designing System with Much Higher Reliability

Now suppose we want to meet the reliability goal of  $1 - 10^{-9}$ .

We decide to use a nonreconfigurable 5-plex (5MR) and build a processor with a failure rate of  $10^{-5}/\text{hour}$ . The probability of system failure is plotted as a function of  $1-C$ :



- The value of  $C$  must now be greater than 0.99999982.
- It can be shown that over a million fault injections would be required to measure this parameter even if we are very optimistic about the testing process
- If each injection required 1 minute, this would require almost 1.9 years of non-stop fault injections.

## A better Way—via Design For Validation

It would be nice if we could design our system so that such an experiment is unnecessary.

- The system is designed so that a single point failure **cannot** cause system failure (i.e.  $C = 1$ ).
- This is demonstrated to be true by formal proof.
- Thus, one uses the power of analysis to eliminate fault-injection style testing.

## WHY FORMAL METHODS?

The successful engineering of complex computing systems will require the application of *mathematically based analysis* analogous to the structural analysis performed before a bridge or airplane wing is built.

## Draft Interim Defence Standard 00-55

Quote from the foreward to the Draft Standard:

The Steering Group "has determined that the current approach which is based on system testing and oversight of the design process will, in the long-term, become cumbersome and inefficient for the assurance of the safety of increasingly sophisticated software".

"The Steering Group therefore proposes the adoption of *Formal Design Methods*, based on rigorous mathematical principles, for the implementation of safety-critical computer software".

## Levels of Formal Methods

Level 0: Static Code Analysis (i.e. no semantic analysis)

Level 1: Specification using mathematical logic or language  
with a formal semantics (i.e. meaning expressible in logic)

Level 2: Formal Specification + Hand Proofs

Level 3: Formal Specification + Mechanical Proofs



## Summary

- Testing and design-diversity techniques inadequate.
- A design-for-validation based on formal methods is needed for the digital flight control systems problem.
- Formal methods will play a major role in the development of future high reliability digital systems.

165  
56-08  
319687  
P-30  
**N91-17565**

## **What FM can offer DFCS Design**

John Rushby

Computer Science Laboratory  
SRI International

PART I - 101

## Overview

- What has actually gone wrong in practice?
- What is the pattern?
- What is the solution?

## **Advanced Fighter Technology Integration (AFTI) F16**

- Triplex DFCS to provide two-fail operative design
- Analog backup
- Digital computers were not synchronized
- "General Dynamics believed synchronization would introduce a single-point failure caused by EMI and lightning effects"

## **AFTI F16 DFCS Redundancy Management**

- Each computer samples sensors independently, uses average of the good channels, with wide threshold
- Single output channel selected from among the good channels
- Output threshold 15% plus rate of change
- Four bad values in a row and the channel is voted out

## **AFTI F16 Flight Test, Flight 15**

- Stores Management System (SMS) relays pilot requests for mode changes to DFCS
- An unknown failure in the SMS caused it to request mode changes 50 times a second
- DFCS responded at a rate of 5 mode changes per second
- Pilot said aircraft felt like it was in turbulence
- Analysis showed that if aircraft had been maneuvering at the time, DFCS would have failed

## **AFTI F16 Flight Test, Flight 36**

- Control law problem led to "departure" of three seconds duration
- Sideslip exceeded  $20^\circ$ , normal acceleration exceeded  $-4g$ , then  $+7g$ , angle of attack went to  $-10^\circ$ , then  $+20^\circ$ , aircraft rolled  $360^\circ$ , vertical tail exceeded design load, failure indications from canard hydraulics, and air data sensor
- Side air data probe blanked by canard at high AOA
- Wide threshold passed error, different channels took different paths through control laws
- Analysis showed this would cause complete failure of DFCS and reversion to analog backup for several areas of flight envelope

## **AFTI F16 Flight Test, Flight 44**

- Asynchronous operation, skew, and sensor noise led each channel to declare the others failed
- Analog backup not selected (simultaneous failure of two channels not anticipated)
- Aircraft flown home on a single digital channel
- No hardware failures had occurred



## **AFTI F16 Flight Test**

- Repeated channel failure indication in flight was traced to roll-axis software switch
- Sensor noise and asynchronous operation caused one channel to take a different path through the control laws
- Decided to vote the software switch
- Extensive simulation and testing performed
- Next flight, same problem still there
- Found that although switch value was voted, the unvoted value was used

## **X29 Flight Test**

- Three sources of air data on X29A: nose and two side probes
- If value from nose is within threshold of both side probes, use nose probe value
- Threshold is large due to position errors in certain flight modes
- If nose probe failed to zero at low speed it would still be within threshold of correct readings
- Aircraft would become unstable and "depart"
- Caught in simulation but 162 flights had been at risk

## **HiMAT Flight Test**

- Single failure in redundant uplink hardware
- Software detected this, and continued operation
- But would not allow the landing skids to be deployed
- Aircraft landed with skid retracted, sustained little damage
- Traced to timing change in the software that had survived extensive testing

## **Gripen Fight Test, Flight 6**

- Unstable aircraft
- Triplex DFCS with Triplex analog backup
- Yaw oscillations observed on several flights
- Final flight had uncontrollable pitch oscillations
- Crashed on landing, broke left main gear, flipped
- Traced to control laws

## Space

- Voyager computer clocks skipped 8 seconds at Jupiter due to high radiation levels (AW&ST Aug 7, 1989)
- So "continuous resynchronization" provided at Neptune
- Also, remember STS-1: "The bug heard round the world" (SEN Oct 1981)

## **FDIR and Crew Interface**

- Imaginary crash scenario
- Broken fan blade on port engine
- Port vibration sensor saturates, limiter cuts in
- Vibration travels down wing, shakes starboard engine
- Starboard vibration sensor reports the attenuated vibration
- Only starboard vibration warning light comes on in cockpit
- Pilot shuts down the good engine, crashes short of runway
- Similar to British Midland 737 crash in 1989

## **Complexity and Integration**

- "The FMS of the A320 'was still revealing software bugs until mid-January,' according to Gérard Guyot (Airbus test and development director). There was no particular type of bug in any particular function, he says. 'We just had a lot of flying to do in order to check it all out. Then suddenly it was working,' he says with a grin" (Flight International, 27 Feb 1989)
- The ATF hardware is ready to go, but cannot be flown because the software engineers "can't get all the 0's and 1's in the right order" (Northrop Engineer, 7 Aug, 1990)

## Complexity and Integration

As of early 1988	A300	A310	A320
Put in service	1982	1983	1988
Number in service	16	149	3
Flight Hours	16,000	810,000	2,000
	Computers		
Autopilot	2 FCC	2 FCC	2 FMGC
Rudder	2 FAC	2 FAC	2 FAC
Autothrottle	1 TCC	1 or 2 TCC	
Slats and flaps		2 SFCC	2 SFCC
Elevator/aileron		2 EFCU	2 ELAC
Spoilers		2 FLC	3 SEC
Fuel management		2 CGCC	
Instruments		3 SGU	3 DMC
Brakes			2 BSCU
Engines		2 FADEC	2 FADEC



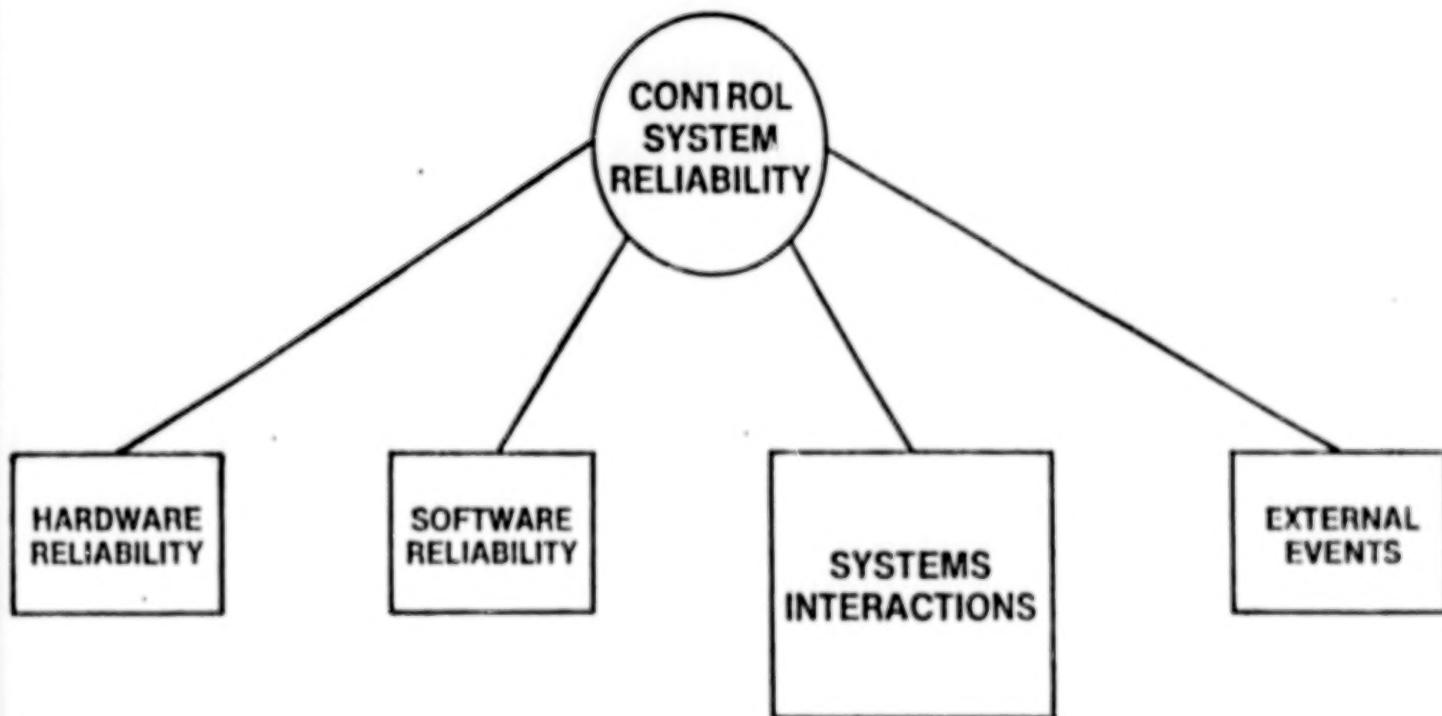
## **Analog, Mechanical Backups**

- Do mechanical and analog backups reduce the requirement for ultra-reliability in DFCS?
- Not if the DFCS is providing stability augmentation or envelope protection
- Similar problem in ATC—potential to move traffic at higher rates than the backup can handle
- No FAA certification credit for mechanical rudder and trim-tab on A320

**Analysis: Dale Mackall, NASA Engineer**  
**AFTI F16 Flight Test**

- Nearly all failure indications were not due to actual hardware failures, but to design oversights concerning asynchronous computer operation
- Failures due to lack of understanding of interactions among
  - Air data system
  - Redundancy management software
  - Flight control laws

# FLIGHT CONTROL SYSTEM RELIABILITY HEAVILY DEPENDENT ON SYSTEM INTERACTIONS



## **Analysis: NASA-LaRC 1988 FCDS Technology Workshop**

- Lack of fully effective design and validation methods with support tools to enable engineering of highly-integrated, flight-critical digital systems
- Complexity of failure containment, test coverage, FMEA, redundancy management, especially in the face of increased integration of flight-critical functions
- Sources of failure:
  - Multiple independent faults (never observed)
  - Single point failures (observed sometimes)
  - Domino failures (most common?)

## **Analysis: Scientific Foundations**

- It is time to place the development of real-time systems on a firm scientific basis. Real-time systems are built one way or another because that was the way the last one was built. And, since the last one worked, we hope that the next one will. (Fred Schneider)
- "Not far from there (CNRS-LAAS), Airbus Industries builds the Airbus A320s. These are the first commercial aircraft controlled solely by a fault-tolerant, diverse computing system. *Strangely enough this development owes little to academia.* (IEEE Micro, April 1989, p6)

## **Analysis**

- The problems of DFCS are the problems of systems whose complexity has exceeded the reach of the intellectual tools employed
- Intuition, experience, and techniques derived from mechanical and analog systems are insufficient for complex, integrated, digital systems

## **Synthesis**

- Computer science has been addressing issues of systematic design, fault tolerance, and the mastery of complexity with some (limited) success for the last 20 years
- But there has been little interest in learning about, and applying this knowledge to, real-time control systems in general (and little opportunity to apply it to DFCS)
- And little of the lore and wisdom of practical real-time control system design has been captured and analyzed

## **What Computer Science Can Offer DFCS**

- Systematic techniques for the construction of trustworthy software, including:
  - Techniques for the precise specification of requirements and the development of designs
  - Systematic approaches to the design and structuring of distributed and concurrent systems
  - Fault tolerant algorithms
  - Systematic methods of testing and analytic methods of verification
- Where do formal methods come in?



## **Applied Mathematics and Engineering**

- Established engineering disciplines use applied mathematics
  - As a *notation* for describing systems
  - As an analytical tool for calculating and *predicting* the behavior of systems
- Computers can provide speed and accuracy for the calculations

## Applied Mathematics and Software Engineering

- The applied mathematics of software is formal logic
- Formal Logic can provide
  - A notation for describing software designs—*formal specification*
  - A calculus for analyzing and predicting the behavior of systems—*formal verification*
- Computers can provide speed and accuracy for the calculations
- Calculating the behavior of software is an exercise in formal reasoning—i.e., theorem proving

## Formal Methods

- Methodologies for using mathematics in software engineering
- Can be applied at many different levels, for both description and analysis
  0. No application of formal methods
  1. Quasi-formal pencil and paper techniques
  2. Mechanized quasi-formal methods
  3. Fully formal pencil and paper techniques
  4. Mechanically checked fully formal techniques

## **Benefits of Formal Specification**

- Unambiguous description facilitates communication among engineers
- Early detection of certain errors
- Encourages systematic, thoughtful approach, reuse of well-understood concepts
- As documentation, reduces some of the difficulties in maintenance and modification

## **Benefits of Formal Verification**

- Subjects the system to extreme scrutiny, increasing designers' understanding of their own creation
- Helps identify assumptions, increases confidence
- Encourages simple, direct designs, austere requirements—better systems
- Encourages and supports a systematic, derivational approach to system design
- Complements testing and allows it to focus on fundamentals

## **Conclusion: What FM Can Offer DFCS**

- Precise *notations* for specifying requirements and designs
- *Concepts* and *structure* for systematic design
- Intellectual tools for *analyzing* the consistency of specifications and the conformance of designs
- A way to regain *intellectual mastery* of complex systems and their interactions

## **Recommendations**

- Just adding formal methods to existing practice is inappropriate
- Capture and analyze lore and wisdom (and mistakes) of actual DFCS designs
- Apply modern Computer Science (including Formal Methods) to develop building blocks for principled DFCS design
- Ultimately, build one and fly it!

195  
57-08  
N91-17566  
57683  
736

## What Can Formal Methods Offer to Digital Flight Control Systems Design?

Formal Methods Workshop  
NASA Langley Research Center  
Hampton, VA.

August 20-23, 1990

Donald I. Good  
Computational Logic, Inc.

### Abstract

Formal methods research is beginning to produce methods which will enable mathematical modeling of the physical behavior of digital hardware and software systems. The development of these methods directly supports the NASA mission of increasing the scope and effectiveness of flight system modeling capabilities.

The conventional, continuous mathematics that is used extensively in modeling flight systems is not adequate for accurate modeling of digital systems. Therefore, the current practice of digital flight control system design has not had the benefits of extensive mathematical modeling which are common in other parts of flight system engineering.

Formal methods research is showing that by using discrete mathematics, very accurate modeling of digital systems is possible. These discrete modeling methods are still in an embryonic stage. But when they are fully developed, they will bring the traditional benefits of modeling to digital hardware and software design. Sound reasoning about accurate mathematical models of flight control systems can be an important part of reducing the risks of unsafe flight control.



# **What Can Formal Methods Offer to Digital Flight Control Systems Design?**

**Donald I. Good**

**Computational Logic, Inc.  
1717 West Sixth, Suite 290  
Austin, Texas 78703**

**512-322-9951**

**good@cli.com**

**"Formal Methods" Enable  
Mathematical Modeling  
of  
Digital Systems  
(Hardware and Software)**

**NASA Mission Objective: Increase the scope and effectiveness of flight system modeling capabilities. -- Lee Holcomb, NASA HQ, 1990.**

# Why Model?

For either design of a new system or operation of an old one, modeling provides...

**Benefits: early error detection**

- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

**Risks: model misrepresents system**

- Inaccurate
- Incomplete

**Kinds of models: physical, analog, schematic, mathematical.**

**Blanchard and Fabrycky. Systems Engineering and Analysis, Prentice Hall, 1990.**

## **Why a Mathematical Model?**

- **High abstraction**
- **High precision**
- **Simulate by manipulating symbols**
- **Represent large classes of system states**
- **Use mathematical deduction**

**Get a lot of system simulation for a little symbol manipulation.**

# Operational Safety

Operating a system safely requires

- accurate predictions

of how it will behave.

Accurate predictions can be obtained from

- sound deductions about
- accurate mathematical models

of system behavior.

## A Classic Model

### Free Fall Distance:

$$f(b,t) = [g(b) * t^{**2}] / 2$$

$g(b)$  = if  $b = \text{"earth"}$  then 32  
          else if  $b = \text{"moon"}$  then ...

$t$  is time (sec)

$f(b,t)$  is distance (ft)

### Simulation:

$$\begin{aligned} f(\text{"earth"}, .7) &= [32 * .7^{**2}] / 2 \\ &= 16 * .49 \\ &= 7.84 \text{ ft} \end{aligned}$$

## Power of Mathematical Deduction

Suppose  $0 \leq t_0 \leq t_1$ .

$t$  in  $[t_0..t_1]$

$f(\text{"earth"}, t)$  in  $(32 * [t_0..t_1]**2) / 2$

$f(\text{"earth"}, t)$  in  $16 * [t_0..t_1]**2$

$f(\text{"earth"}, t)$  in  $16 * [t_0**2..t_1**2]$

(\*\* is monotonic)

**Physical simulation of this result is impossible because  $[t_0..t_1]$  contains an infinite number of values.**



## Validating a Model

- Ultimately, the accuracy of a model of a physical system must be validated by testing it against measured, observed behavior of the actual physical system.
- One cannot construct a mathematical proof that a model is an accurate representation of a physical system.
- Typically, one iterates through a process of
  - stating a mathematical model
  - testing it against physical observations
  - adjusting the model

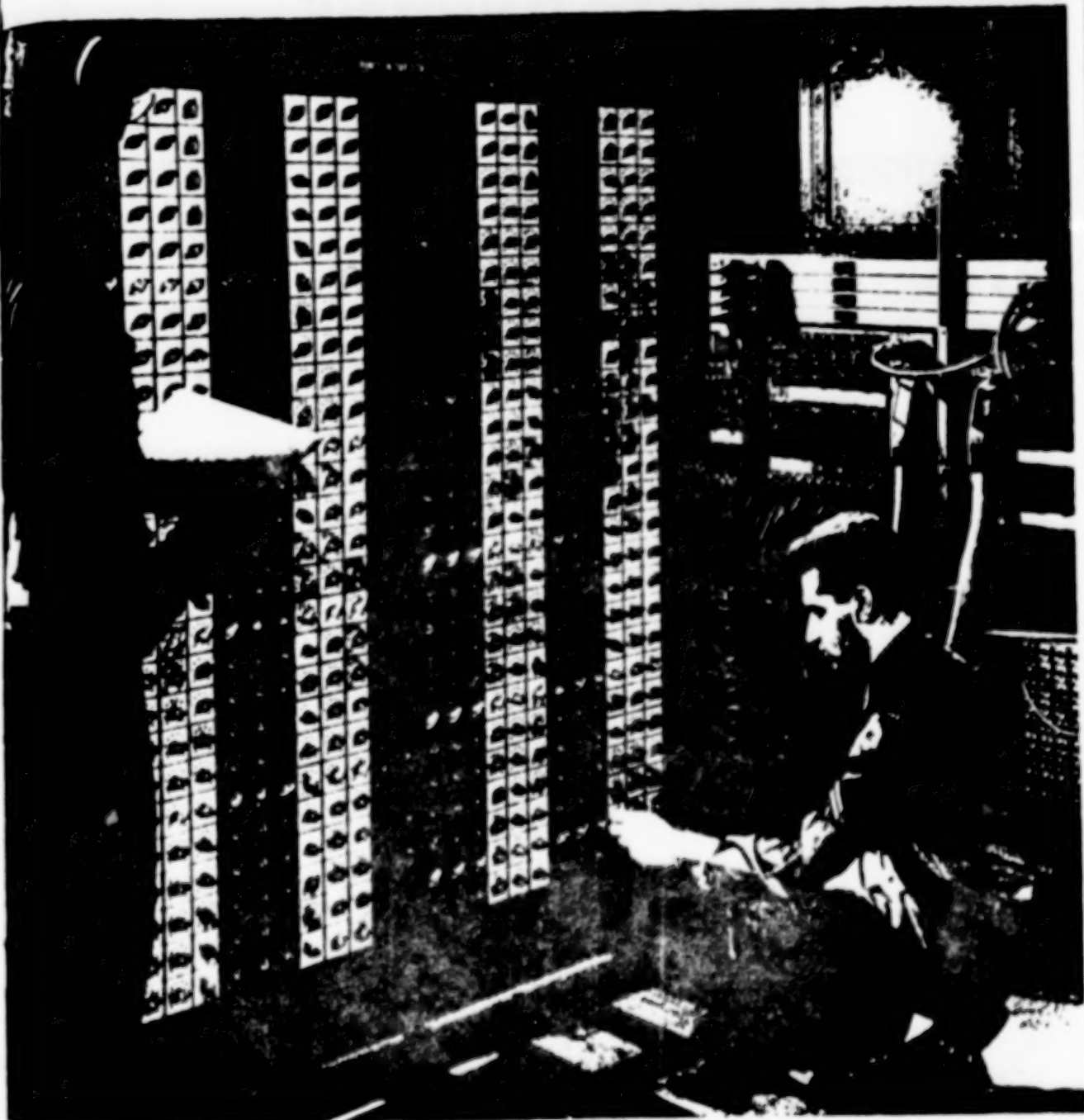


## **Hardware Model Observables**

**A hardware system  
is composed  
of physical switches.**

**Nancy Stern. From ENIAC to UNIVAC: An  
Appraisal of the Eckert-Mauchly Computers.  
Digital Equipment Corporation, 1981.**

**Next page.**



The switches of Function Table A are being set by Plc. Homer Spence and Lt. pl. Irwin Goldstein. Three manually set function tables served as read-only memory units. Courtesy Moore School of Electrical Engineering, University of Pennsylvania.

ORIGINAL PAGE  
BLACK AND WHITE PHOTOGRAPH

ORIGINAL PAGE IS  
OF POOR QUALITY

## Use Discrete Mathematics to Model Hardware

- Switches by binary digits
- Operation by recursive functions

s0    -----  
      | 0 1 1 0 0 0 0 1 1 1 1 |  
      -----

s1    -----  
      | 1 0 1 0 0 1 1 0 0 0 0 |  
      -----

s2    -----  
      | 1 1 1 0 0 0 1 0 1 0 1 |  
      -----

o o o

## An MC68020 Machine Model

```
MC68020(s,n) =  
  if haltp(s) or n=0  
  then s  
  else MC68020(NEXT(s), n-1)  
  
NEXT(s) =  
  if evenp(pc(s))  
  then if pc_readp(mem(s),pc(s))  
        then EXECUTE(FETCH(pc(s),s),  
                      update_pc(s,...))  
        else halt(s, pc_signal)  
  else halt(s,pc_odd_signal)  
  
EXECUTE(ins,s) =  
... [50 pages for 90% user ins.] ...
```

**Provides a mathematically precise and consistent machine language reference manual.**

**Yuan Yu. PhD Thesis (in progress). University of Texas.**

# **The VIPER Machine**

**A 32-bit microprocessor "whose functions are totally predictable."**

- Accumulator
- 2 index registers
- Program counter
- Comparison register
- 16 instructions

**Avra Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. Technical Report 104, University of Cambridge Computer Laboratory, January, 1987.**

**W. J. Cullyer. Implementing High Integrity Systems: The VIPER Microprocessor. In Computer Assurance, COMPASS 88. IEEE, June, 1988.**

## A VIPER Machine Model

```
NEXT (ram, p, a, x, y, b, stop) =  
  if stop  
  then (ram, p, a, x, y, b, stop)  
  else (noinc \/\ illegaladdr) \/  
        if (illegalcl \/\ illegalsp)  
          \/\ (illegalonp \/\ illegalwr)  
        then (ram, newp, a, x, y, b, T)  
        else ... [about 7 pages] ...
```

where

ram - a memory of 32-bit words  
p - 20-bit program counter  
a - 32-bit accumulator  
x, y - 32-bit index registers  
b - 1 bit compare result register  
stop - stop flag



# **The FM8502 Machine**

**A 32-bit microprocessor.**

- **2 address architecture**
- **4 addressing modes**
- **8 general purpose registers**
- **2<sup>19</sup> 20-bit instructions**

**Warren A. Hunt, Jr. FM8501: A Verified Microprocessor, Ph.D. Thesis, The University of Texas at Austin, 1985.**

**-----, Microprocessor Design Verification. Journal of Automated Reasoning. Vol. 5, No. 4, Dec 1989.**

## An FM8502 Machine Model

```
FM8502 (ms, mn) =  
  if not (listp(mn))  
  then ms  
  else FM8502 (NEXT (ms),  
               rest (mn))
```

```
NEXT (ms) =  
  list (next_memory      (ms),  
        next_register_file (ms),  
        next_carry_flag   (ms),  
        next_overflow_flag (ms),  
        next_zero_flag    (ms),  
        next_negative_flag (ms) )
```

... [about 10 pages] ...



## An FM8502 Register Transfer Model

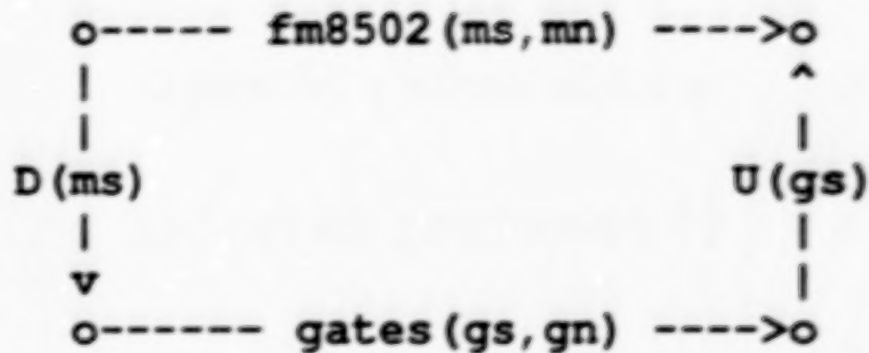
```
GATES (gs, gn) =  
  if not (listp(gn))  
  then gs  
  else GATES (COMB_LOGIC (gs, car(gn)),  
               cdr(gn))
```

```
COMB_LOGIC (gs, gn) =  
... [on bit operators, e.g., b_xor] ...
```

where

gs	- [regs, flags, mem, int-regs]
regs	- 8 32-bit vectors
flags	- 4 Booleans
mem	- $2^{32}$ 32-bit vectors
int-regs	- 32-bit vectors for internal registers, flags, latches

## Connecting the Models



Theorem:  $H(ms, mn) \rightarrow$   
 $fm8502(ms, mn) =$   
 $U(gates(D(ms), Kg(ms, mn, md)))$

Under the conditions  $H$ ,

- the  $fm8502$  model is just as accurate as  $gates$
- but with some details suppressed by  $U$ .

## **Software Model Observables**

**Programming languages provide  
a wide variety of ways  
of describing them, but  
the observables are still switches,  
and so are programs!**

## Models of Programmed Machines

- A machine is programmed by setting the switches which it will interpret as instructions during its operation. (Before stored-program machines, this process was called "setting up" the machine.)

```
-----  
| 0 1 1 0 0 0 0 1 1 1 1 |  
-----  
|   prog   |   data   |
```

- These switches are the program. They control the subsequent operation of the machine.
- A computer program is a physical control mechanism.
- The bit string "011000" is a mathematical description of the control mechanism.

## **A Model of a Programmed Machine**

**A model of machine  $M$  operating on initial state  $s_0$  for  $k(s_0)$  steps under the control of the program described by  $p_0$  is given by**

$$M(s_0, k(s_0))$$

**where**

$s_0$  - a machine state such that  
 $\text{prog}(s_0) = p_0$

$\text{prog}(s)$  - a function that extracts the  
program description from  $s$

### **Operating Requirements**

**A model of a machine programmed to satisfy an  
operating requirement  $R(s_0, sk)$  is given by**

$$R(s_0, M(s_0, k(s_0)))$$

000B	000D	0002	08B8	000E	0003	004B	0003	00BF	000E	0CDD	004D	0002	0009	0041	0002
000F	10CB	0002	0000	31CB	0002	0000	12CB	0002	000D	13CB	0002	000E	0CCB	0002	000A
00CB	0002	0005	08CB	0002	0006	0FCB	0002	0007	0041	0002	0008	50CB	0002	0000	104B
0002	000D	104B	0003	000E	0000	084B	0003	0002	004D	0002	0009	0041	0002	000F	004D
0003	0002	0041	0003	009F	00DA	0003	01DE	0003	084B	0003	0002	0041	0003	0008	18B8
0000	0002	39B8	0000	0002	1AB8	0000	0003	F84B	0007	0002	D84B	0006	0002	884B	000F
0002	984B	0004	0002	784B	0003	0002	584B	0002	0002	0000	000E	09F3	004B	0003	000B
000E	0CDD	000E	0CCA	000E	0CA8	0002	0C86	000E	09F3	004B	0003	00BF	000E	0CDD	000E
0CCA	004D	0002	0002	0041	0002	00D3	00CB	0002	0001	01CB	0002	0000	0002	0C86	000E
09F3	0009	0008	0004	0083	0008	0000	000A	0A98	0083	0008	0001	000A	0AFD	0083	0008
0002	000A	0B6A	0002	08A5	084B	0006	0002	0049	0006	0010	084B	0007	0003	004B	0003
00BF	000E	0CDD	08B8	000C	0002	084B	0004	000E	004D	0004	0008	084B	0003	0002	004D
0003	000E	0841	0003	0004	0041	0003	01F3	000E	0CE1	000A	0AE7	084B	0002	0007	000E
0CA8	084B	0003	0006	004D	0003	0002	0041	0003	00D3	00C3	0003	0003	0006	0C96	11C3
0003	000C	0006	0C96	00CB	0003	0000	01CB	0003	0000	084B	0002	0006	004B	0003	00BF
000E	0CA8	0002	0C96	004B	0003	00BF	000E	0CCA	104B	0003	000C	004D	0003	0002	0041
0003	0003	00CB	0003	0002	09CB	0003	0006	0002	0C86	084B	0006	0002	0049	0006	0010
004B	0003	00BF	000E	0CDD	08B8	000C	0002	084B	0004	0002	004D	0004	0008	084B	0003
0006	004D	0003	0080	0841	0003	0004	0041	0003	01F3	000E	0CDD	000A	0854	000E	0CDD
000E	0CCA	104B	0003	000C	004D	0003	0009	0041	0003	000F	00CB	0003	0002	084B	0003
0006	004D	0003	0002	0041	0003	00D3	00C3	0003	0002	0006	0C96	11C3	0003	000C	0006
0C96	00CB	0003	0000	01CB	0003	0000	084B	0002	0006	004B	0003	00BF	000E	0CA8	0002
0C96	004B	0003	00BF	000E	0CCA	104B	0003	000C	004D	0003	0002	0041	0003	00D3	00CB
0003	0003	09CB	0003	0006	0002	0C86	084B	0007	0003	004B	0003	00BF	000E	0CDD	08B8
000C	0002	084B	0003	0002	004D	0003	0008	0041	0003	0173	000E	0CE1	000A	088F	084B
0002	0007	000E	0CA8	0006	000C	0006	0C96	00A6	000C	0002	0C96	004B	0003	00BF	000E
0CCA	104B	0003	000C	004D	0003	0002	0041	0003	00D3	00CB	0003	0004	01CB	0003	0000
00C2	0C86	004B	0003	00BF	000E	0CDD	08B8	000C	0002	084B	0003	0002	004D	0003	0008
0041	0003</														

08/27/90

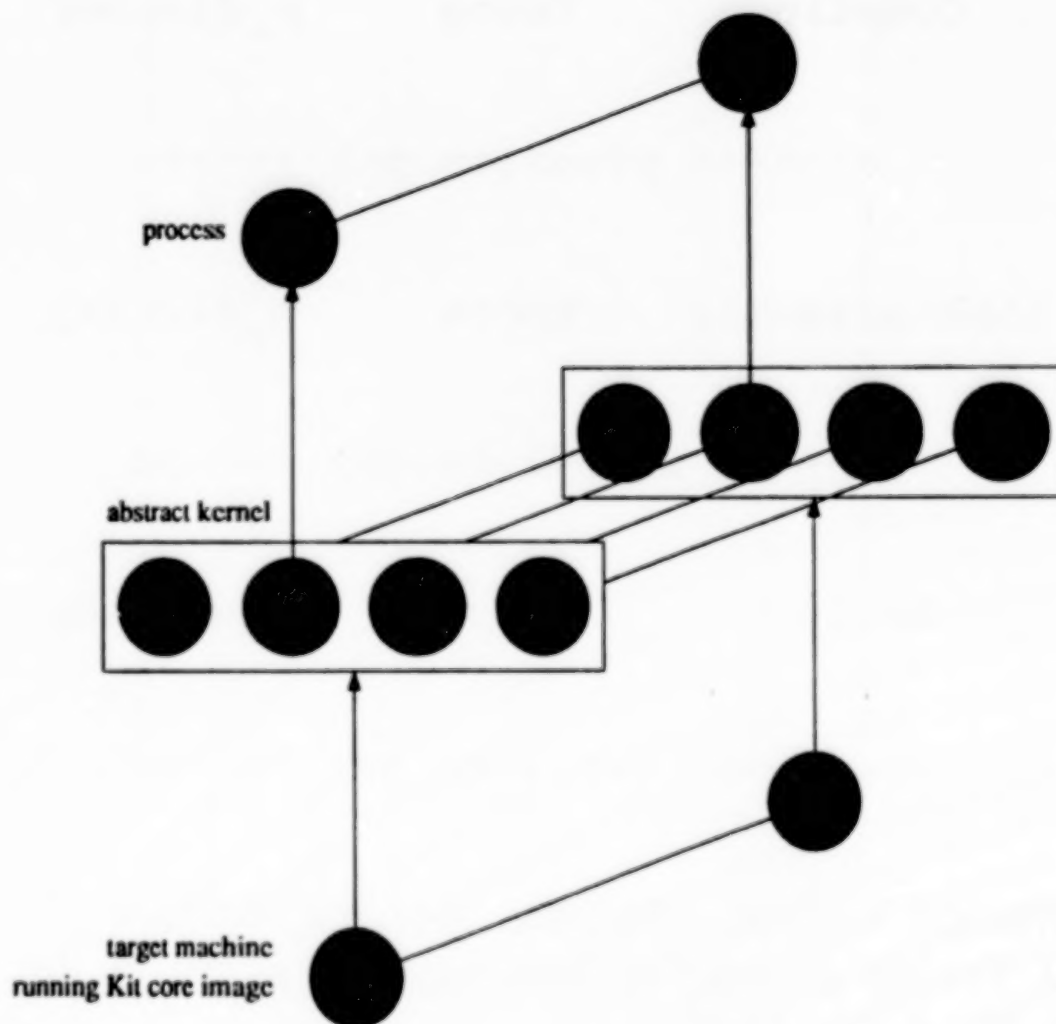


# **The Kit Separation Kernel**

- **Uses a modified FM8501 (ms, mn) machine**
- **Interrupts for timer and I/O**
- **Process management**
  - **fixed number of processes**
  - **process scheduling (round robin)**
  - **process communication (message passing)**
  - **response to error conditions**
- **Device management for character I/O to asynchronous devices**
- **Memory management uses hardware protection**

**William R. Bevier. Kit: A Study in Operating System Verification. IEEE Transactions on Software Engineering. November 1989.**

## Kit Operating Requirement, R





# The CLInc Stack

```

      o-- uGypsy (yx, yp, yd, yn) ->o
      |                               ^
      |                               |
Compile      Young      p_display
      |                               |
      v                               |
      o----- piton (ps, pn) ----->o
      |                               ^
      |                               |
Link-assemble  Moore    m_display
      |                               |
      v                               |
      o----- fm8502 (ms, mn) ----->o
      |                               ^
      |                               |
Reify         Hunt      g_display
      |                               |
      v                               |
      o----- gates (gs, gn) ----->o

```

Warren A. Hunt, J Strother Moore II, William D. Young. Journal of Automated Reasoning. Vol. 5, No. 4, Dec 1989.

**Warren A. Hunt, J Strother Moore II, William D. Young. Journal of Automated Reasoning. Vol. 5, No. 4, Dec 1989.**

# **The Piton Language**

**The Piton language has**

- **execute-only program space**
- **read/write global arrays**
- **recursive subroutine calls**
- **formal parameters**
- **user-visible stack**
- **stack-based instructions**
- **flow-of-control instructions.**

**The cross assembler produces an FM8502 binary core image.**

## **The Micro Gypsy Language**

**The Micro Gypsy subset of Gypsy has**

- **types integer, boolean, character**
- **one dimensional arrays**
- **procedure calls with pass by reference parameters**
- **sequential control structures if, loop,**
- **condition handling signal..when.**

**The compiler produces Piton.**

## The Stack Theorem

Theorem:  $H'(yx, yp, yd, yn) \rightarrow$   
 $uGypsy(yx, yp, yd, yn) =$   
 $U'(gates(D'(yx, yp, yd),$   
 $Kg'(yx, yp, yd, yn, md)))$

Proof: Mechanically checked.

Under the conditions  $H'$ ,

- the  $uGypsy$  model is just as accurate as gates
- but with many details suppressed by  $U'$ .

## Boyer-Moore Logic

Robert S. Boyer, J Strother Moore II. A Computational Logic Handbook, Academic Press, 1988.

Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. TR 19, Computational Logic, Inc., 1988.

## **A Hierarchy of Models of a Programmed Machine**

**R(yx0, yp0, yd0, ydk)**

**uGypsy(yx0, yp0, yd0, yk(yx0, yp0, yd0))**

**piton(ps0, pk(ps0))**

**fm8502(ms0, mk(ms0))**

**gates(gs0, gk(gs0))**

**Corresponding to these is a hierarchy of program  
descriptions....**

## Operating Requirement

```
procedure mult (var ans:fm8502_int;  
                i,j:fm8502_int) =  
begin  
  ENTRY j ge 0;  
  EXIT  ans = NTIMES(i,j);  
    pending;  
end;
```

```
type fm8502_int =  
  integer[-(2**31)..(2**31)-1];
```

{A Simple Problem Domain Theory}

```
function NTIMES(x,y:integer):integer =  
begin  
  exit (assume result =  
    if y = 0 then 0  
    else if y = 1 then x  
      else x + NTIMES(x,y-1)  
    fi    fi);  
end;
```

## Gypsy Program Description

```
procedure mult(var ans:fm8502_int;  
               i,j:fm8502_int) =  
begin  
  ENTRY j ge 0;  
  EXIT  ans = NTIMES(i,j);  
  var k:fm8502_int := 0;  
    k := j;  
  ans := 0;  
  loop  
    ASSERT j ge 0 & k in [0..j]  
      & ans = NTIMES(i,j-k);  
    if k le 0 then leave end;  
    ans := ans + i;  
    k := k - 1;  
  end;  
end;
```



## Piton Program Description

```
(MG-MULT
  (K ZERO ONE B ANS I J) ;formals
  NIL                     ;locals
  (PUSH-LOCAL ANS)        ;ans := 0;
  (PUSH-CONSTANT (INT 0))
  (CALL MG-SIMPLE-CONSTANT-ASSIGNMENT)
  (PUSH-LOCAL K)          ;k := j;
  (PUSH-LOCAL J)
  (CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
  (DL L-1 NIL (NO-OP))    ;loop
  (PUSH-LOCAL B)          ; b := k le 0
  (PUSH-LOCAL K)
  (PUSH-LOCAL ZERO)
  (CALL MG-INTEGER-LE)
  (PUSH-LOCAL B)          ; if b then leave
  (FETCH-TEMP-STK)
  (TEST-BOOL-AND-JUMP FALSE L-3)
  (PUSH-CONSTANT (NAT 0))
  (POP-GLOBAL C-C)
  (JUMP L-2)
  (JUMP L-4)
  (DL L-3 NIL (NO-OP))
  (DL L-4 NIL (NO-OP))
  (PUSH-LOCAL ANS)        ; ans := ans + i;
  (PUSH-LOCAL ANS)
  (PUSH-LOCAL I)
  (CALL MG-INTEGER-ADD)
  (PUSH-GLOBAL C-C)
  ... [14 more support routines] ...
```



## FM8502 Program Description

[illegible]

... [10 more pages] ... 11

## Mathematical Requirements

- **Unambiguous:** Requirements have a well-defined interpretation that tells exactly what they do say.
- **Analyzable:** Do the requirements say the "right" thing?

$R(x, y) \rightarrow \text{good\_thing}(x, y)$

- **Consistency:** Requirements contain no contradictions.
- **Enable modeling a program component before building it (and thereby save the time and cost of designing a poor program.)**

To get these benefits, the requirements notation must have a rigorous mathematical foundation (semantics).

## Design >> Requirements

- There is more to designing a digital system than just stating and refining mathematical requirements.
- One must still construct a program for some machine.
- Mathematical models of commonly used languages and machines are still very scarce.

## **Summary**

**For either design of a new system or operation of an old one, mathematical modeling of digital flight control systems offers**

**Benefits: early error detection**

- **Saves time**
- **Saves money**
- **Saves operational disruption**
- **Saves operational mishaps**

**Risks: model misrepresents system**

- **Inaccurate**
- **Incomplete**

## Conventional Non-Wisdom

Use "formal methods" (mathematical modeling)

- only after a system is built to certify it
- only before a system is built to design it
- to guarantee perfect system behavior
- to eliminate the need for testing

**High Level Design Proof of a  
Reliable Computing Platform**

Ben L. Di Vito  
Ricky W. Butler  
James L. Caldwell

NASA Langley Research Center  
Hampton, VA 23665

N91-17567

232  
58-60

319689

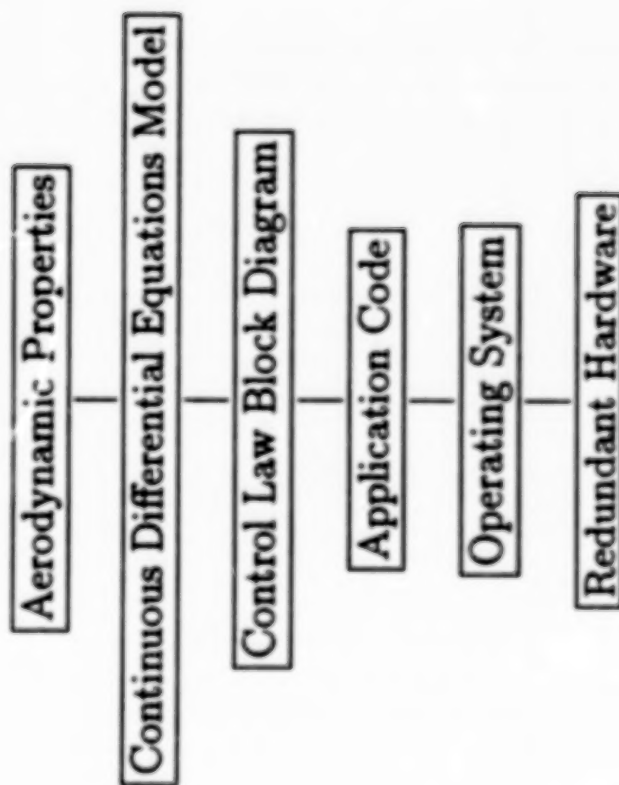
228

## Outline

- Research Objectives
- Reliable Computing Platform
- High-Level Design Specifications
- Correctness Proofs
- Voting Patterns

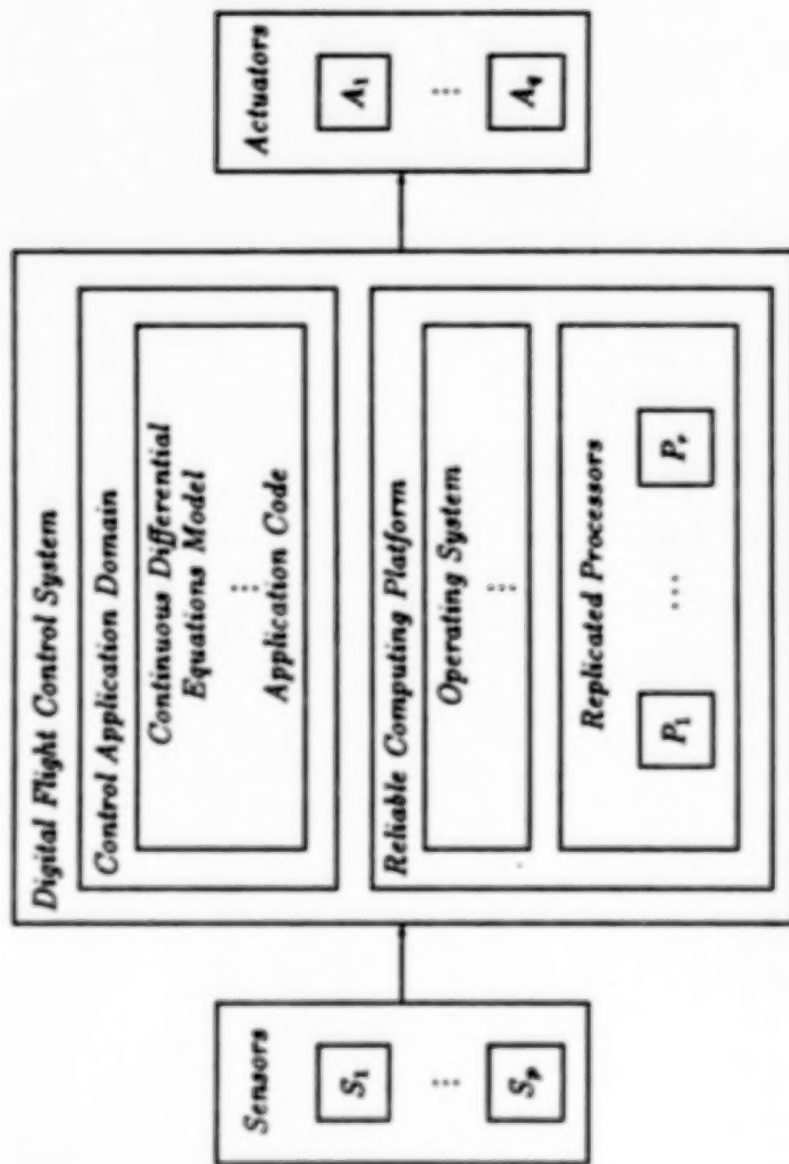


## Digital Flight Control Systems





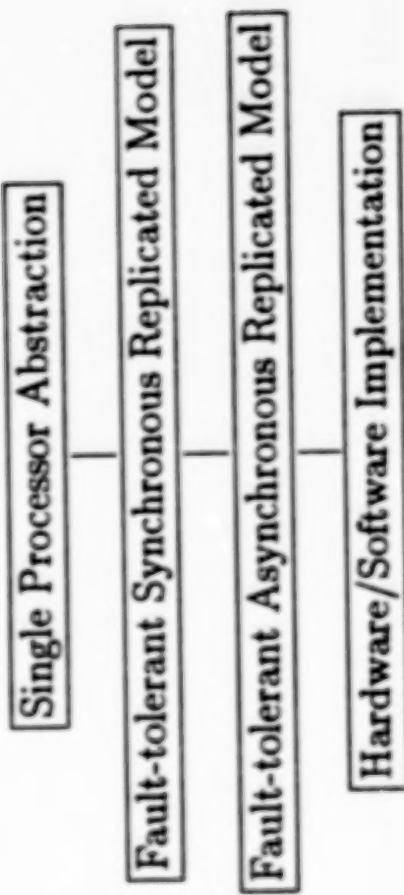
# Reliable Computing Platform



## Research Objectives

- Establish hardware/software platform for ultra-reliable computing
- Use fault-tolerant computer architecture
- Use formal methods to prevent design and implementation errors
  - first specify in conventional mathematical notation
  - then specify and mechanically verify in EHD M
- Construct reliability model to quantify reliability estimate

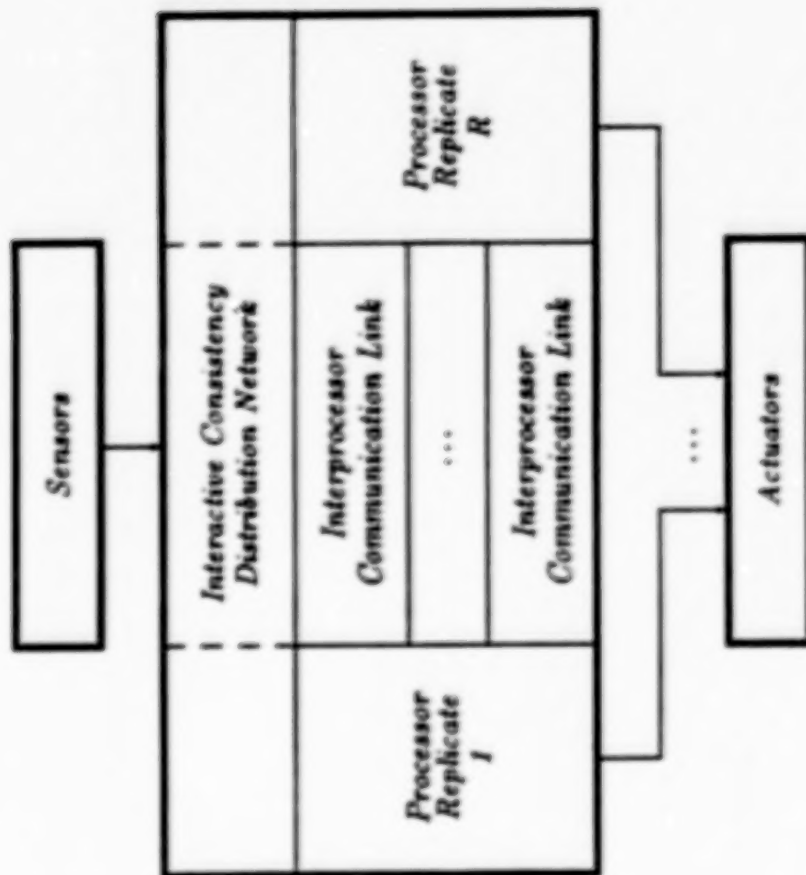
# Operating System for Control Applications



## Application Task Characteristics

1. Fixed set of tasks
2. Hard deadlines
3. Multi-rate cyclic scheduling
4. Minimal jitter
5. Upper bound on task execution time
6. Precedence constraints

# Architectural Concept

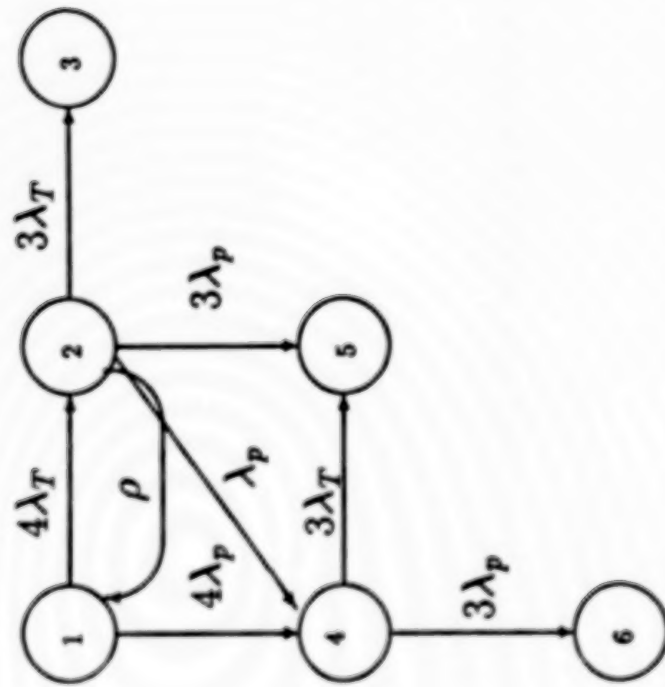


## Design Decisions

1. the system is non-reconfigurable
2. the system is frame-synchronous
3. the scheduling is static, non-preemptive
4. internal voting is used which can recover the state of a processor affected by a transient fault within a bounded time

## Reliability Modeling

Reliability model of quadruplex version of system

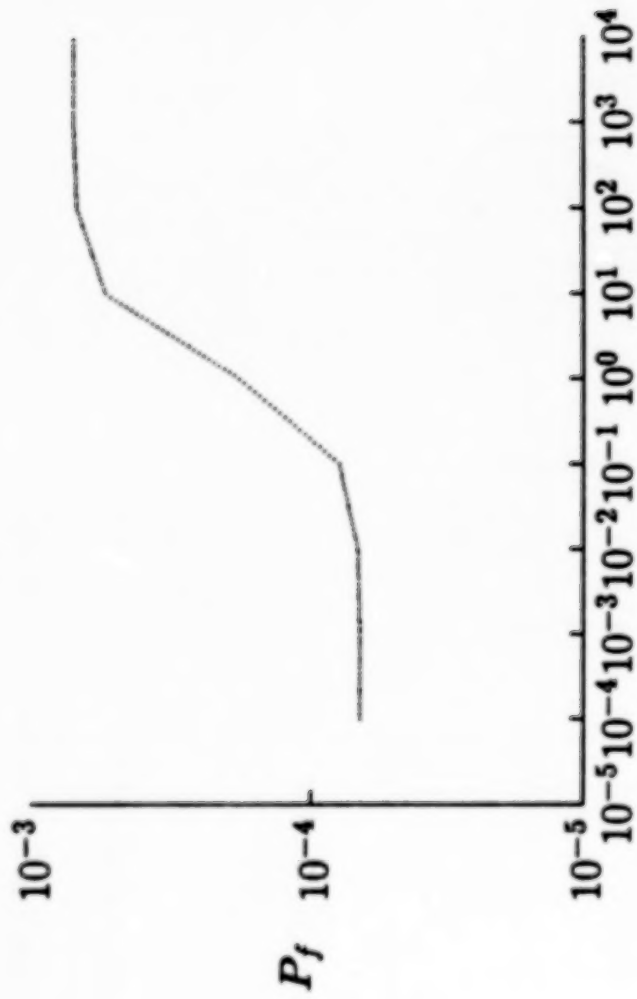


$\lambda_T$  = transient fault rate ( $\sim 10^{-3}/\text{hr}$ )

$\lambda_P$  = permanent fault rate ( $\sim 10^{-4}/\text{hr}$ )

$\rho$  = rate of recovery from transient fault (design-dependent)

## Transient Fault Recovery

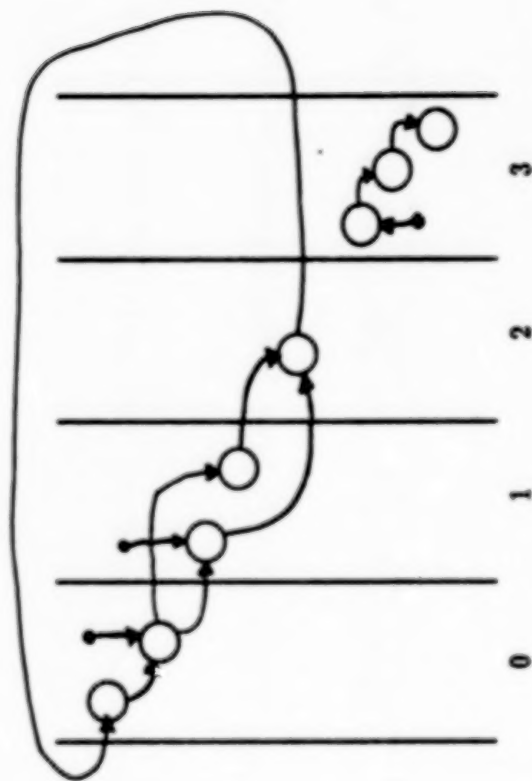


Mean Time to Recover From Transient (hours)

Note inflection point on the order of one minute



# Application Definition



$M$  frames = 1 cycle

$M_i > 0$  subframes per frame

$K$  tasks

$(i, j)$  = cell (frame, subframe)

ST: scheduled task for cell  $(i, j)$

TI: task inputs for cell  $(i, j)$  {tasks have no permanent state}

AO: actuator output tasks

IR: initial task inputs

# Task Schedule

Frame	0	T1	T2	T3	T1	T4	
	1	T5	T1	T2	T6	T1	
	.						
	.						
	.						
	M-1	T1	T9	T8	T4	T7	

## Uniprocessor Model

State of abstract machine given by:

$$OS\_state = ( \quad frame : \{0..M-1\}, \\ \quad results : \{0..M-1\} \times nat \rightarrow D )$$

The OS state transition is defined by the function  $OS$ .

$$OS : Sin \times OS\_state \rightarrow OS\_state$$

$$OS(s, u) = (u.frame \oplus 1, \lambda i, j. new\_results(s, u, i, j))$$

where

$$x \oplus y = (x + y) \bmod M$$

$$x \ominus y = (x + M - y) \bmod M$$

$$new\_results(s, u, i, j) = \text{if } i = u.frame \\ \text{then } exec(s, u, i, j) \\ \text{else } u.results(i, j)$$

## Uniprocessor Model (Cont'd.)

$$exec : Sin \times OS\_state \times \{0..M-1\} \times nat \rightarrow D$$

$$exec(s, u, i, j) = f_{ST(i,j)}(arg(TI(i,j)[1], s, u, i, j), \dots, arg(TI(i,j)[n], s, u, i, j))$$

$$arg : triple \times Sin \times OS\_state \times \{0..M-1\} \times nat \rightarrow D$$

$$arg(t, s, u, i, j) = \text{if } t.type = sensor$$

$$\text{then } s[t.i]$$

$$\text{else if } t.i = i \wedge t.j < j$$

$$\text{then } exec(s, u, i, t.j)$$

$$\text{else } u.results(t.i, t.j)$$

Actuator output is a function of the OS state:

$$UA(u) = [\sum_{k=1}^q Act(u, k)]$$

$$Act(u, k) = \begin{cases} u.results(u.frame \ominus 1, j) & \text{if } \exists j : AO(u.frame \ominus 1, j) = k \\ \phi & \text{otherwise} \end{cases}$$

## Replicated Processor Model

The replicated processor model is based on a replicated state and transitions that allow for faults in the replicates

$$Repl : ICin \times Repl\_state \times fault\_status \rightarrow Repl\_state$$

$$Repl(c, r, \Phi) = [\substack{R \\ k=1} RT(c, r, k, \Phi)]$$

$$RT(c, r, k, \Phi) = \text{if } \Phi[k] \text{ then } \perp \text{ else } (frame\_vote(r, \Phi), Repl\_results(c, r, k, \Phi))$$

$$frame\_vote(r, \Phi) = maj([\substack{R \\ l=1} FV_l])$$

where  $FV_l = \text{if } \Phi[l] \text{ then } \perp \text{ else } r[l].frame \oplus 1$

$$maj : sequence(D \cup \{\perp\}) \rightarrow D \cup \{\perp\}$$

## Replicated Processor Model (Cont'd.)

$$VP : \{0..M-1\} \times nat \times \{0..M-1\} \rightarrow \{T, F\}$$

$VP(i, j, n) = T$  iff we are to vote  $OS.results(i, j)$  during frame  $n$ .

$$Repl.results(c, r, k, \Phi) =$$

$$\lambda i, j. \text{ if } VP(i, j, r[k].frame)$$

$$\text{ then } results\_vote(c, r, i, j, \Phi)$$

$$\text{ else } new\_results(c[k], r[k], i, j)$$

$$results\_vote(c, r, i, j, \Phi) = maj([\![_{l=1}^R RV_l])$$

where  $RV_l = \text{if } \Phi[l] \text{ then } \perp \text{ else } new\_results(c[l], r[l], i, j)$ .

Replicated actuator output considers fault status indicators:

$$RA : Repl\_state \times fault\_status \rightarrow RAout$$

$$RA(r, \Phi) = [\![_{k=1}^R RA_k]$$

where  $RA_k = \text{if } \Phi[k] \text{ then } \perp \text{ else } UA(r[k])$

### A Simple Fault Model

[illegible]

The results we seek must hold for all  $\mathcal{F} : \{1..R\} \times \text{nat} \rightarrow \{T, F\}$  that satisfy a condition for maximally unfortunate fault behavior. Define a *working* processor as follows.

$$W: \{1..R\} \times nat \times fault\_fn \rightarrow \{T, F\}$$

$$\mathcal{W}(k, n, \mathcal{F}) = \forall j: 0 \leq j \leq \min(n, N_R) \supset \sim \mathcal{F}(k, n - j)$$

A processor that is nonfaulty, but not yet working, is considered to be *recovering*. The number of working processors is given by:

$$\omega(n, \mathcal{F}) = |\{k \mid W(k, n, \mathcal{F})\}|$$

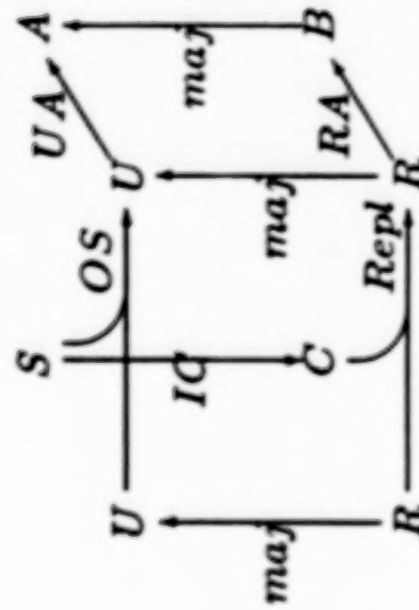
**Definition 1** *The Maximum Fault Assumption for a given fault function  $\mathcal{F}$  is that  $\omega(n, \mathcal{F}) > R/2$  for every frame  $n$ .*

All theorems about state machine correctness are predicated on this assumption.

## Framework For Proving State Machine Correctness

Functions needed to bridge the gap between the two machines are those that do the following:

1. Map sensor inputs for  $UM$  into replicated sensor inputs for  $RM$ .
2. Map replicated actuator outputs from  $RM$  into actuator outputs for  $UM$ .
3. Map replicated OS states of  $RM$  into uniprocessor OS states of  $UM$ .





## Correctness Criteria

**Definition 2** *RM* correctly implements *UM* under assumption  $\mathcal{P}$  iff the following formula holds:

$$\forall \mathcal{F} : \mathcal{P}(\mathcal{F}) \supset \exists S, \forall n > 0 : a_n = \nu(b_n) \quad (1)$$

where  $a_n$  and  $b_n$  can be characterized as functions of an initial state and all prior inputs.

We parameterize the concept of necessary assumptions using the predicate  $\mathcal{P}$ . For the replicated system, it will be instantiated by the Maximum Fault Assumption:

$$\mathcal{P}(\mathcal{F}) = (\forall m : \omega(m, \mathcal{F}) > R/2).$$

## Derived Correctness Criteria

**Definition 3 (Replicated OS Correctness Criteria)** *RM* correctly implements *UM* if the following conditions hold:

- (1)  $u_0 = \text{maj}(r_0)$
- (2)  $\forall \mathcal{F}, (\forall m : \omega(m, \mathcal{F}) > R/2) \supset$   
 $\forall S, \forall n > 0 : OS(s_n, \text{maj}(r_{n-1})) = \text{maj}(\text{Repl}(IC(s_n), r_{n-1}, \mathcal{F}_n^R))$
- (3)  $\forall \mathcal{F}, (\forall m : \omega(m, \mathcal{F}) > R/2) \supset$   
 $\forall S, \forall n > 0 : UA(\text{maj}(r_n)) = \text{maj}(RA(r_n, \mathcal{F}_n^R))$

# Sufficient Conditions for Correctness

Generic State Machine Correctness Criteria

↑↑

Replicated OS Correctness Criteria

↑↑

Consensus Property

↑↑

Replicated State Invariant

↑↑

Full Recovery Property

↑↑

Voting Pattern

## Intermediate Assertions

**Definition 4 (Consensus Property)** *For  $\mathcal{F}$  satisfying the Maximum Fault Assumption, the assertion*

$$\mathcal{W}(p, n-1, \mathcal{F}) \supset r_{n-1}[p] = \text{maj}(r_{n-1}) \wedge r_n[p] = \text{maj}(r_n)$$

*holds for all  $p$  and all  $n > 0$ .*

**Definition 5 (Replicated State Invariant)** *For fault function  $\mathcal{F}$  satisfying the Maximum Fault Assumption, the following assertion is true for every frame  $n$ :*

$$\begin{aligned} & (n = 0 \vee \sim \mathcal{F}(p, n-1)) \supset \\ & \quad r_n[p].\text{frame} = \text{maj}(r_n).\text{frame} = n \bmod M \wedge \\ & \quad (\forall i, j : \text{rec}(i, j, \mathcal{L}(p, n, \mathcal{F}), \mathcal{H}(p, n, \mathcal{F}), T) \supset \\ & \quad \quad r_n[p].\text{results}(i, j) = \text{maj}(r_n).\text{results}(i, j)). \end{aligned}$$

## Recovery Concepts

Recovery of state element  $(i, j)$  where last faulty frame was  $f$  and processor has been healthy for  $h$  frames:

$$\begin{aligned} \text{rec}(i, j, f, h, e) = & \text{if } h \leq 1 \text{ then } F \\ & \text{else } (VP(i, j, f \oplus h) \wedge e) \vee \\ & \quad \text{if } i = f \oplus h \\ & \quad \quad \text{then } \bigwedge_{l=1}^{|TI(i,j)|} RI(TI(i, j)[l], i, j, f, h) \\ & \quad \quad \text{else } \text{rec}(i, j, f, h - 1, T) \end{aligned}$$

$$\begin{aligned} RI(t, i, j, f, h) = & (t.type = \text{sensor}) \vee \\ & \text{if } t.i = f \oplus h \wedge t.j < j \\ & \quad \text{then } \text{rec}(t.i, t.j, f, h, F) \\ & \quad \text{else } \text{rec}(t.i, t.j, f, h - 1, T) \end{aligned}$$

**Definition 6 (Full Recovery Property)** The predicate  $\text{rec}(i, j, f, N_R, T)$  holds for all  $i, j, f$ .

## Continuous Voting

$$VP(i, j, k) = T \quad \forall i, j, k$$

$$N_R = 2 \quad (\text{Actual } N_R = 1)$$

- Specifies that the *entire state* will be voted every frame
- Not very practical
- But proof is simple

## Cyclic Voting

$$VP(i, j, k) = (i = k) \quad \forall i, j, k$$

$$N_R = M + 1.$$

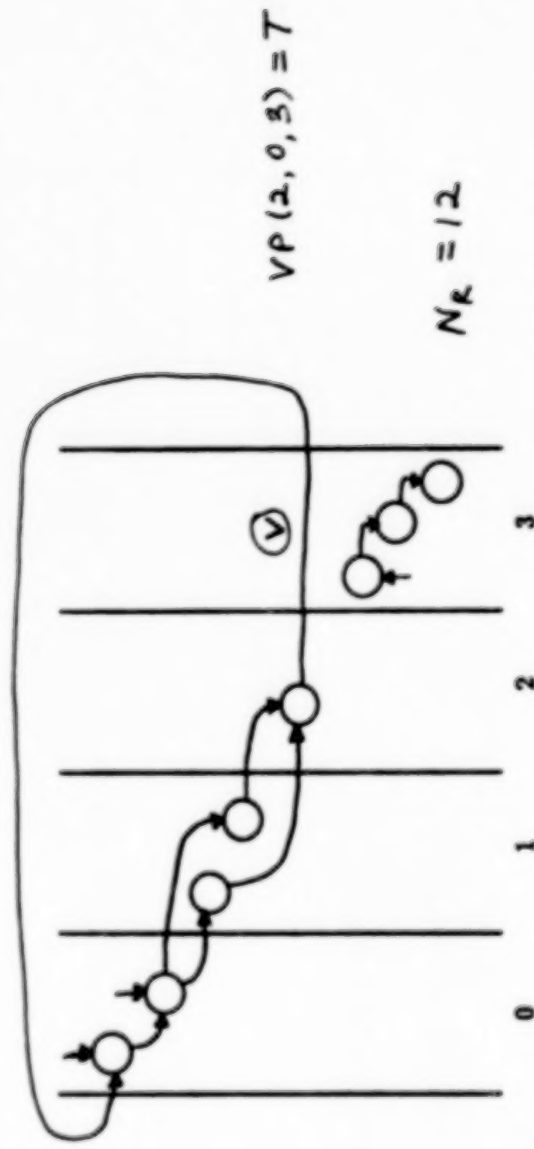
- Only results just computed will be voted in a frame
- More practical
- Proof almost as simple

Frame

0	1	2	3	0	1	2	3
v				v			
	v				v		
		v				v	
			v				v

Portion voted

## Minimal Voting



- Vote only portion of state that will not be recovered from new sensor values.
- Construct  $VP$  to ensure each cycle of graph is cut by at least one vote.
- $N_R = L_C + L_N + M$  where
  - $L_C$  = maximum frame length for all cycles
  - $L_N$  = maximum frame length for all noncyclic paths



## Summary

- Ultra-reliable control systems hard to achieve
- Simple fault-tolerant design postulated
- Formal specification of design constructed
- Preliminary correctness proofs obtained
- Will extend from here
  - more sophisticated designs
  - mechanical verification

**A Verified Model of Fault-Tolerance**

John Rushby

Computer Science Laboratory  
SRI International

N91-17568

266  
Sg-4

319690-

220

## **Transient Faults are Common and Important**

NASA-LaRC 1988 FCDS Technology Workshop:

- A number of DFCS are highly susceptible to radiated EM energy (composite materials provide less shielding, densely packed VLSI more susceptible to SEU)
- Designers must *prove* that their design will always recover from any and all non-hard faults reasonably quickly

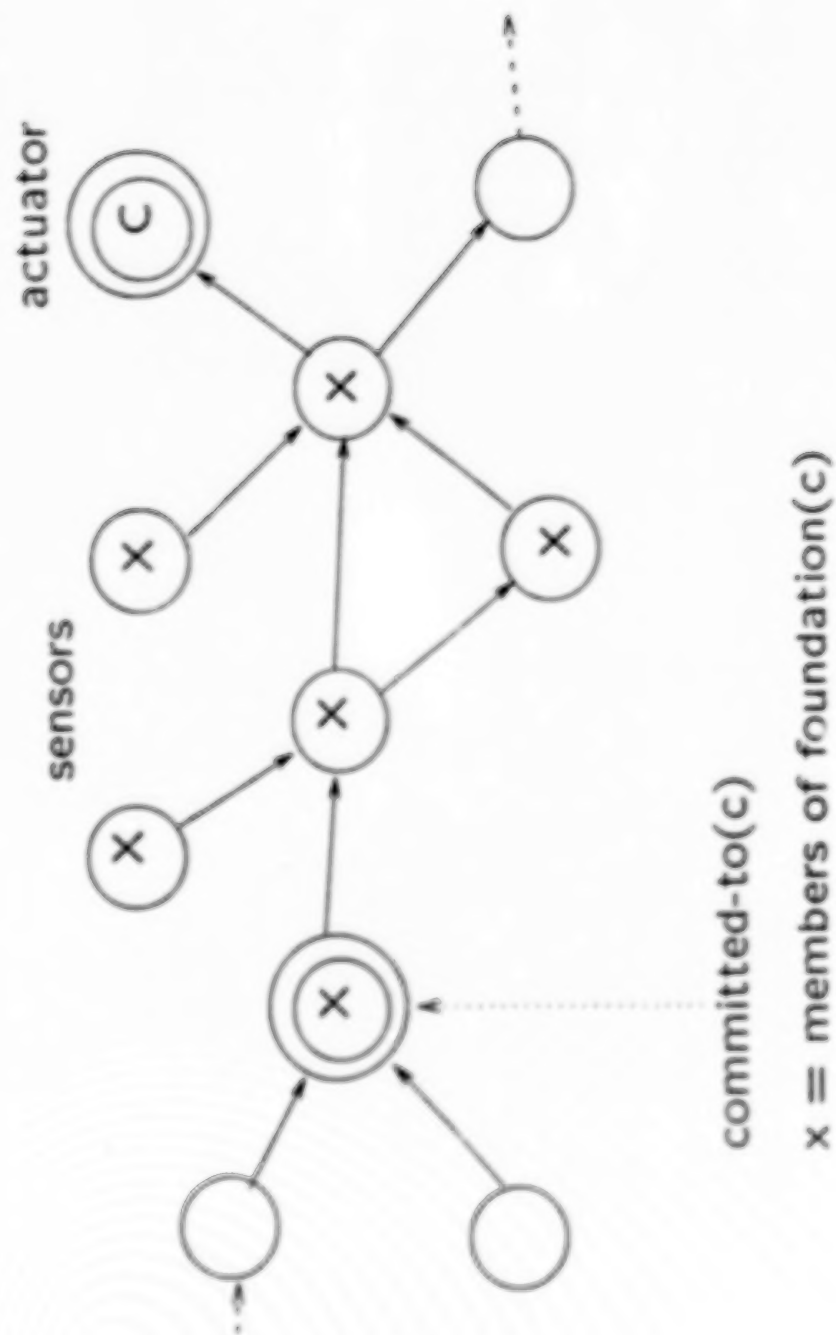
## Goal

- A model of a replicated system with exact-match voting
- A fault model that includes transients
- A theorem that establishes the conditions under which the system provides fault tolerance
- A formal specification of the model and a mechanically checked verification of the theorem that is consonant with the journal-level presentation

## Status

- Model based closely on that developed by Butler, Caldwell, and DeVito at LaRC, but simplified and more abstract
  - Does not model frames and cycles
  - Does not model sensor failure or loss of frame counter
- Model and theorem described in draft journal-level report
- Specification and verification in Ehdm completed (Jim Caldwell provided stimulation and help in the proof)
- Currently reconciling the two
- Next step is to address the (over) simplifications

## General Idea



## Sets

```
sets: MODULE [T: TYPE]
EXPORTING ALL
THEORY
```

```
set: TYPE IS function[T -> bool]
```

```
x, y, z: VAR T
a, b, c: VAR set
```

```
union: function[set, set -> set] ==
(LAMBDA a, b : (LAMBDA x : a(x) OR b(x)))
```

```
subset: function[set, set -> bool] =
(LAMBDA a, b : (FORALL z : a(z) IMPLIES b(z)))
```

```
member: function[T, set -> bool] == (LAMBDA x, b : b(x))
```

```
empty: function[set -> bool] = (LAMBDA a : (FORALL x : NOT a(x)))
```

```
emptyset: set == (LAMBDA x : false)
```

```
fullset: set == (LAMBDA x : true)
```

```
extensionality: AXIOM
```

```
(FORALL x : member(x, a) = member(x, b)) IMPLIES (a = b)
```

## Cardinality

```
m, n, p: VAR nat

card: function[set -> nat]

card_ax: AXIOM
  card(union(a, b)) + card(intersection(a, b)) = card(a) + card(b)

card_subset: AXIOM subset(a, b) IMPLIES card(a) <= card(b)

card_empty: AXIOM card(a) = 0 IFF empty(a)

empty_prop: LEMMA card(a) > 0 IMPLIES (EXISTS x : member(x, a))

card_prop: LEMMA
  subset(a, c)
  AND subset(b, c)
  AND 2 * card(a) > card(c) AND 2 * card(b) > card(c)
  IMPLIES card(intersection(a, b)) > 0
```



## Sensors etc.

```
C: TYPE

a, c: VAR C

cell-types: TYPE = (sensor-cell, actuator-cell, task-cell)

cell-type: function[C -> cell-types]

sensors: TYPE FROM C WITH (LAMBDA c : cell-type(c) = sensor-cell)

actuators: TYPE FROM C WITH (LAMBDA c : cell-type(c) = actuator-cell)

active-tasks: TYPE FROM C WITH
  (LAMBDA c : cell-type(c) /= sensor-cell)

voted: TYPE FROM C

voted_ax: AXIOM
  (c IN actuators IMPLIES c IN voted)
  AND (c IN voted IMPLIES NOT (c IN sensors))

Gbar: function[C, C -> bool]

sensor_ax: AXIOM (EXISTS a : Gbar(a, c)) IFF NOT (c IN sensors)
```

## Simple Machine

$step(\sigma, c, n) = \sigma$  with  $[c := \text{if } c \in C\_S \text{ then } sensor(c)(n) \text{ else } task(c)(\sigma)]$

$run(0) = (\lambda c : \perp)$   
 $run(n+1) = step(run(n), sched(n+1), n+1).$

step: function[state, C, M  $\rightarrow$  state] =  
    (LAMBDA s, c, m : s  
      WITH [c :=  
        IF c IN sensors THEN sensor(c)(m) ELSE task(c)(s) END IF])  
identity: function[M  $\rightarrow$  nat] == (LAMBDA m : m)  
run: RECURSIVE function[M  $\rightarrow$  state] =  
    (LAMBDA m :  
      IF m = 0 THEN undef ELSE step(run(m-1), sched(m), m) END IF)  
    BY identity

## TCC'S

(\* Subtype TCC generated for the first argument to task in dependency \*)

dependency\_TCC1: FORMULA

(c IN active\_tasks AND (FORALL a : Gbar(a, c) IMPLIES s(a) = t(a)))  
IMPLIES (cell\_type(c) /= sensor\_cell)

(\* Subtype TCC generated for the first argument to sensor in step \*)

step\_TCC1: FORMULA (c IN sensors) IMPLIES (cell\_type(c) = sensor\_cell)

(\* Subtype TCC generated for the first argument to task in step \*)

step\_TCC2: FORMULA

(NOT (c IN sensors)) IMPLIES (cell\_type(c) /= sensor\_cell)

(\* Subtype TCC generated for the first argument to run in run \*)

run\_TCC1: FORMULA (m >= 0) IMPLIES (NOT (m = 0)) IMPLIES (m - 1 >= 0)

(\* Termination TCC generated for run \*)

run\_TCC2: FORMULA

(m >= 0) IMPLIES (NOT (m = 0)) IMPLIES identity(m) > identity(m - 1)

## Replicated Machine

$$\neg \mathcal{F}(i)(n) \supset sstep(\rho, c, n)(i) = step(\rho(i), c, n),$$

$$\neg \mathcal{F}(i)(n) \supset vote(\rho, c, n)(i) = \text{if } c \in C\_V \text{ then } \rho(i) \text{ with } [c := maj\{\rho(j)(c) | j \in R\}] \\ \text{else } \rho(i)$$

$$rstep(\rho, c, n) = vote(ssstep(\rho, c, n), c, n).$$

## Replicated Machine

```

sstep_ax: AXIOM
  NOT (F(i)(m)) IMPLIES sstep(rs, c, m)(i) = step(rs(i), c, m)

maj_ax: AXIOM
  (EXISTS A :
    2 * card(A) > card(fullset[R])
    AND (FORALL i : member(i, A) IMPLIES rs(i)(c) = x))
    IMPLIES maj(rs, c) = x

vote_ax: AXIOM
  NOT (F(i)(m))
    IMPLIES vote(rs, c, m)
      = IF c IN voted
        THEN rs WITH [(i)(c) := maj(rs, c)]
        ELSE rs END IF

rstep: function[rstate, C, M -> rstate] ==
  (LAMBDA rs, c, m : vote(sstep(rs, c, m), c, m))

rrun: RECURSIVE function[M -> rstate] =
  (LAMBDA m :
    IF m = 0
      THEN (LAMBDA i : undef)
      ELSE rstep(rrun(m - 1), sched(m), m) END IF)
    BY identity

```

## Foundation etc.

$$foundation(c) = \begin{cases} \{c\} & \text{if } c \in (C_S \cup C_V) \\ \{c\} \cup \bigcup_{(a,c) \in \bar{G}} foundation(a) & \text{otherwise} \end{cases}$$

$$support(c) = \begin{cases} \{c\} \cup \bigcup_{(a,c) \in \bar{G}} foundation(a) & \text{if } c \in C_V \\ foundation(c) & \text{otherwise.} \end{cases}$$

$$committed\text{-}to(c) = \min\{when(a) \mid a \in support(c)\}.$$

## Foundation etc.

```
foundation: RECURSIVE function[C -> set[C]] =
(LAMBDA c :
  (LAMBDA a :
    c = a
    OR (NOT (c IN voted OR c IN sensors)
      AND (EXISTS b :
        Gbar(b, c) AND member(a, foundation(b))))))

BY dowhen

backup: function[C -> set[C]] =
(LAMBDA c :
  (LAMBDA a :
    (EXISTS b : Gbar(b, c) AND member(a, foundation(b))))))

support: function[C -> set[C]] =
(LAMBDA c :
  (LAMBDA a :
    member(a, foundation(c))
    OR (c IN voted AND member(a, backup(c))))))

critical_times: function[C -> set[M]] ==
(LAMBDA c : (LAMBDA t : member(sched(t), support(c))))

committed_to: function[C -> M] == (LAMBDA c : min(critical_times(c)))
```



## OK and MOK

$$OK(i)(c) = (\forall n : committed\_to(c) \leq n \leq when(c) \supset \neg \mathcal{F}(i)(n)).$$

$$MOK(c) = \exists \Theta \subseteq R, |\Theta| > r/2, i \in \Theta \supset OK(i)(c).$$

```

OK: function[R -> set[C]] =
  (LAMBDA i :
    (LAMBDA c :
      (FORALL m :
        committed_to(c) <= m AND m <= dowhen(c)
        IMPLIES NOT F(i)(m))))

working: function[C -> set[R]] == (LAMBDA c : (LAMBDA i : OK(i)(c)))

MOK: function[C -> bool] =
  (LAMBDA c : 2 * card(working(c)) > card(fullset[R]))

```



# Theorem

If

$$\forall a : \text{when}(a) \leq \text{when}(c) \supset \text{MOK}(a),$$

then

$$\forall j : \text{OK}(j)(c) \supset \text{rrunto}(c)(j)(c) = \text{runto}(c)(c).$$

```

safe: RECURSIVE function[C -> bool] =
  (LAMBDA c : MOK(c) AND (FORALL a : Gbar(a, c) IMPLIES safe(a)))
  BY downen

correct: function[C -> bool] =
  (LAMBDA c :
    (FORALL j : OK(j)(c) IMPLIES rrunto(c)(j)(c) = runto(c)(c)))

the_result: THEOREM safe(c) IMPLIES correct(c)

```

## Noetherian Induction

```

noetherian: MODULE [dom: TYPE, <: function[dom, dom -> bool]]
ASSUMING
  measure: VAR function[dom -> nat]
  a, b: VAR dom

  well_founded: FORMULA
    (EXISTS measure : a < b IMPLIES measure(a) < measure(b))
THEORY

  p, A, B: VAR function[dom -> bool]
  d, d1, d2, d3, d4: VAR dom

  general_induction: AXIOM
    (FORALL d1 : (FORALL d2 : d2 < d1 IMPLIES p(d2)) IMPLIES p(d1))
    IMPLIES (FORALL d : p(d))

  mod_induction: THEOREM
    (FORALL d3, d4 : d4 < d3 IMPLIES A(d3) IMPLIES A(d4))
    AND (FORALL d1 :
      (FORALL d2 : d2 < d1 IMPLIES (A(d1) AND B(d2)))
      IMPLIES B(d1))
    IMPLIES (FORALL d : A(d) IMPLIES B(d))

PROOF
  mod_proof: PROVE mod_induction d1 <- d1@p1, d3 <- d1@p1, d4 <- d2
    FROM general_induction p <- (LAMBDA d : A(d) IMPLIES B(d))
END noetherian

```

## The Proof

```
correctness_proof: MODULE
USING correctness, voted_step, nonvoted_step, sensor_step,
  noetherian[C, Gbar]
PROOF
  a, c: VAR C

  discharge_well_founded: PROVE well_founded_measure <- dowhen FROM
    Gbar_when c <- b

  inductive_step: LEMMA
    (FORALL a : Gbar(a, c) IMPLIES safe(c) AND correct(a))
    IMPLIES correct(c)

  almost_final_proof: PROVE inductive_step a <- a@p7 FROM
    sensor_inductive_step, voted_inductive_step, nonvoted_inductive_step,
    induction_body a <- a@p1, induction_body a <- a@p2,
    induction_body a <- a@p3, induction_body

  final_proof: PROVE the_result FROM
    mod_induction A <- safe, B <- correct, d <- c, d2 <- a@p3,
    safe a <- d4@p1, c <- d3@p1,
    inductive_step c <- d1@p1

END correctness_proof
```

## Summary

- Formal specification and verification revealed typos in the original report
- Exposed omission in original proof
- Led to stronger theorem and more elegant proof (using Noetherian rather than ordinary induction)
- Confirmed that Ehdm has the capability to specify interesting and useful properties in a direct, natural, and readable manner
- Proofs were hard (three intensive man-weeks, 92 lemmas); I haven't yet gone back to see why that was so
- We have the beginnings of a formally verified model for a fault tolerant operating system

**The Design and Proof of Correctness  
of a Fault-Tolerant Circuit**

280  
510-23  
319691  
7-15  
**N91-17569**

William R. Bevier  
William D. Young

Computational Logic, Inc.  
1717 W. 6th Street  
Austin, Texas

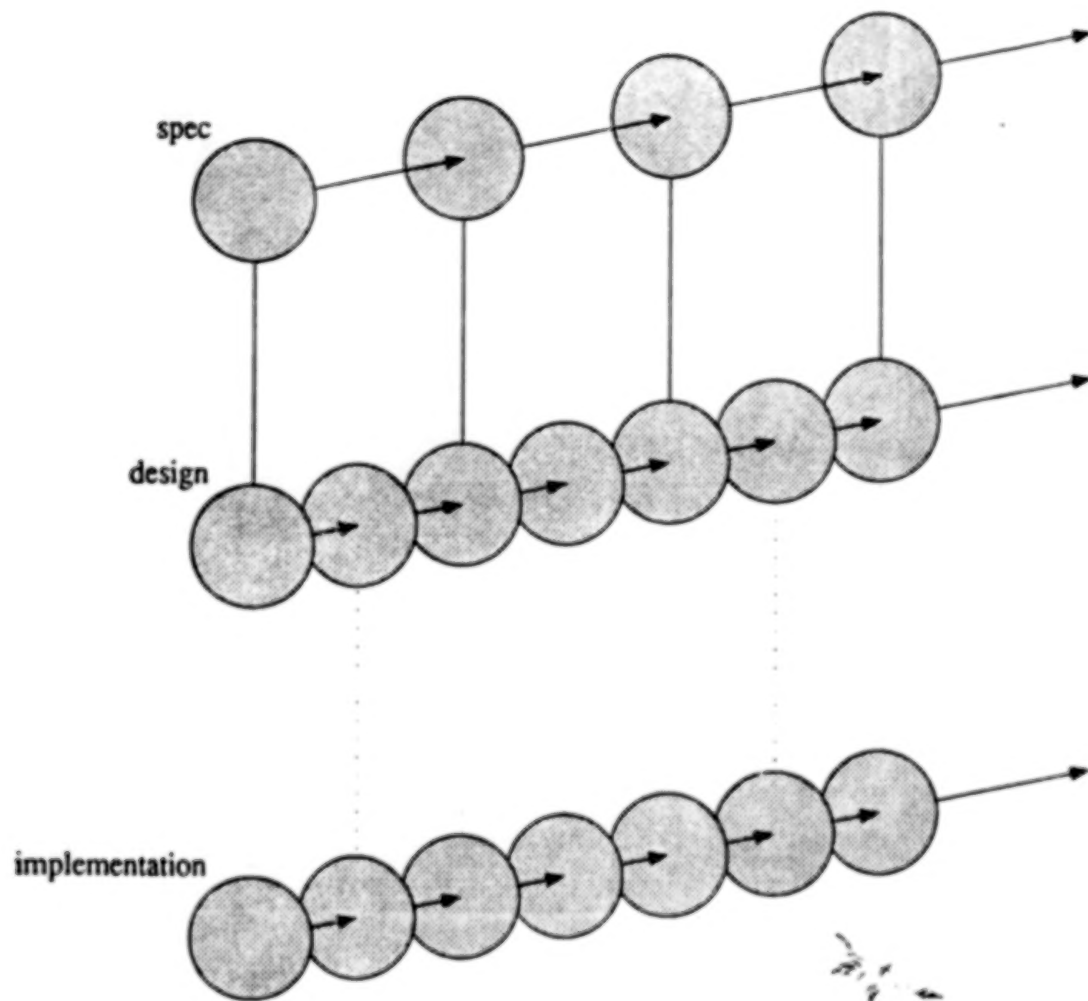
## What We Accomplished

- A formal statement of Interactive Consistency Conditions<sup>1</sup> in the Boyer-Moore logic.
- A formal statement of the Oral Messages algorithm *OM* in the Boyer-Moore logic.
- A mechanically checked proof that *OM* satisfies the Interactive Consistency conditions.
- A mechanically checked proof of the optimality result: no algorithm can tolerate fewer faults than *OM* yet still achieve Interactive Consistency.
- The use of *OM* in a functional specification for a fault-tolerant device.
- A formal description of the design of the device.
- A mechanically checked proof that the device design satisfies the specification.
- An implementation of the design in programmable logic arrays.

---

<sup>1</sup>See "The Byzantine Generals Problem", Lamport, Shostak and Pease, ACM Toplas, Vol 4, No 3, July 1982.

## A Stack of Related Machines



## The Specification

The specification is a function that describes a finite state machine.

At every step, each of  $N$  processes

1. reads its sensor input,
2. exchanges its sensor value with all other processes,
3. produces an *interactive consistency vector* (ICV) that contains what it concludes is each other process's value, and
4. applies a filter function to the ICV to produce an output.



## Properties of the Specification Function

The exchange of sensor values is accomplished by an algorithm called *OM*.

*OM* achieves *interactive consistency*. That is,

A process sends a message to  $n-1$  destination processes.

1. All non-faulty destination processes agree on the same received value.
2. If the sending process is non-faulty, then every non-faulty destination process receives the message sent.

*OM* has been defined as a function in the Boyer-Moore logic, and a proof that interactive consistency is achieved has been mechanically checked.

## Formal Statement of Correctness of *OM*

Let

- $n$  be the number of processes,
- $L$  be the set  $\{0, \dots, n-1\}$ ,
- $g, i, j \in L$  be process names,
- $x$  be  $g$ 's local value, and
- $m$  give the number of rounds of information exchange.

The interactive consistency conditions are stated as follows.

$$\begin{aligned}
 & \neg \text{faulty}(i) \\
 & \& \neg \text{faulty}(j) \\
 & \& 3 \text{faults}(L) < n \\
 & \& \text{faults}(L) \leq m \\
 \rightarrow & OM(n, g, x, m)[i] = OM(n, g, x, m)[j].
 \end{aligned}$$

$$\begin{aligned}
 & \neg \text{faulty}(g) \\
 & \& \neg \text{faulty}(i) \\
 & \& 3 \text{faults}(L) < n \\
 & \& \text{faults}(L) \leq m \\
 \rightarrow & OM(n, g, x, m)[i] = x
 \end{aligned}$$

## Specification Abstraction

The following aspects of the specification are not constrained:

1. The number of processes.
2. The types of the input and output values.
3. The nature of the filter function.

## What Interactive Consistency Guarantees

The specification can be thought of as a function which

- receives a sequence of  $N$ -tuples of input values, and
- produces a sequence of  $N$ -tuples of output values.

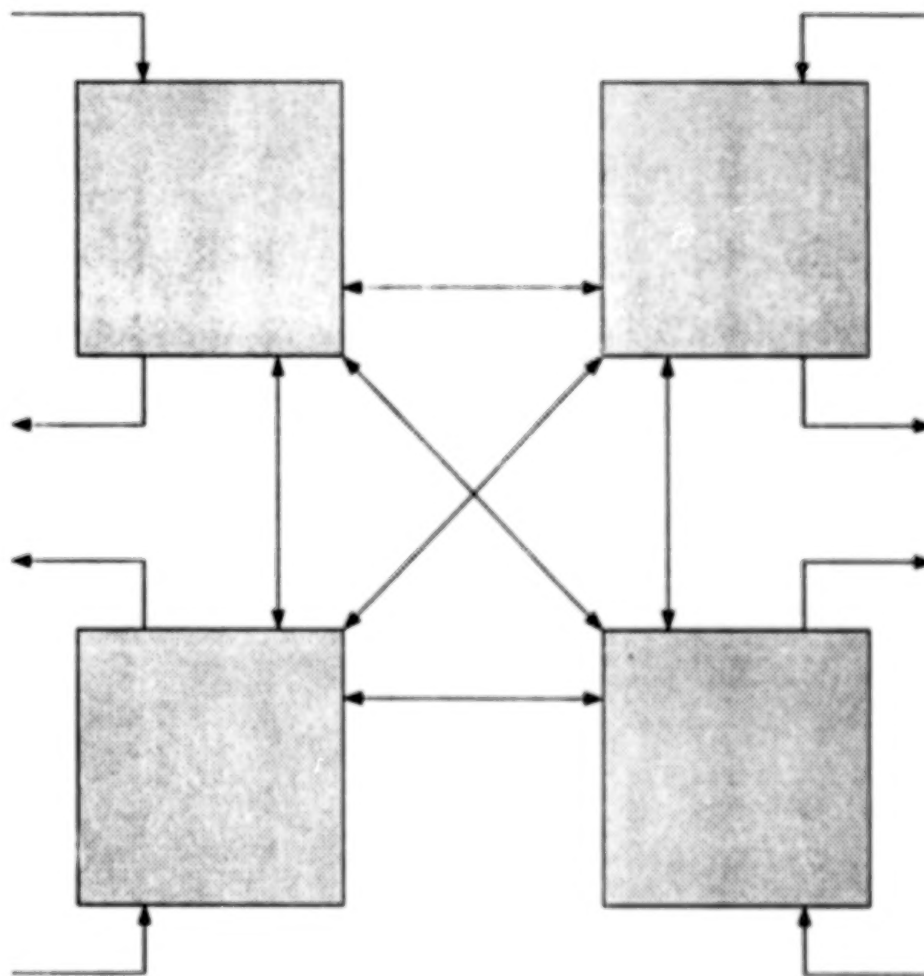


Because of Interactive Consistency, we can conclude:

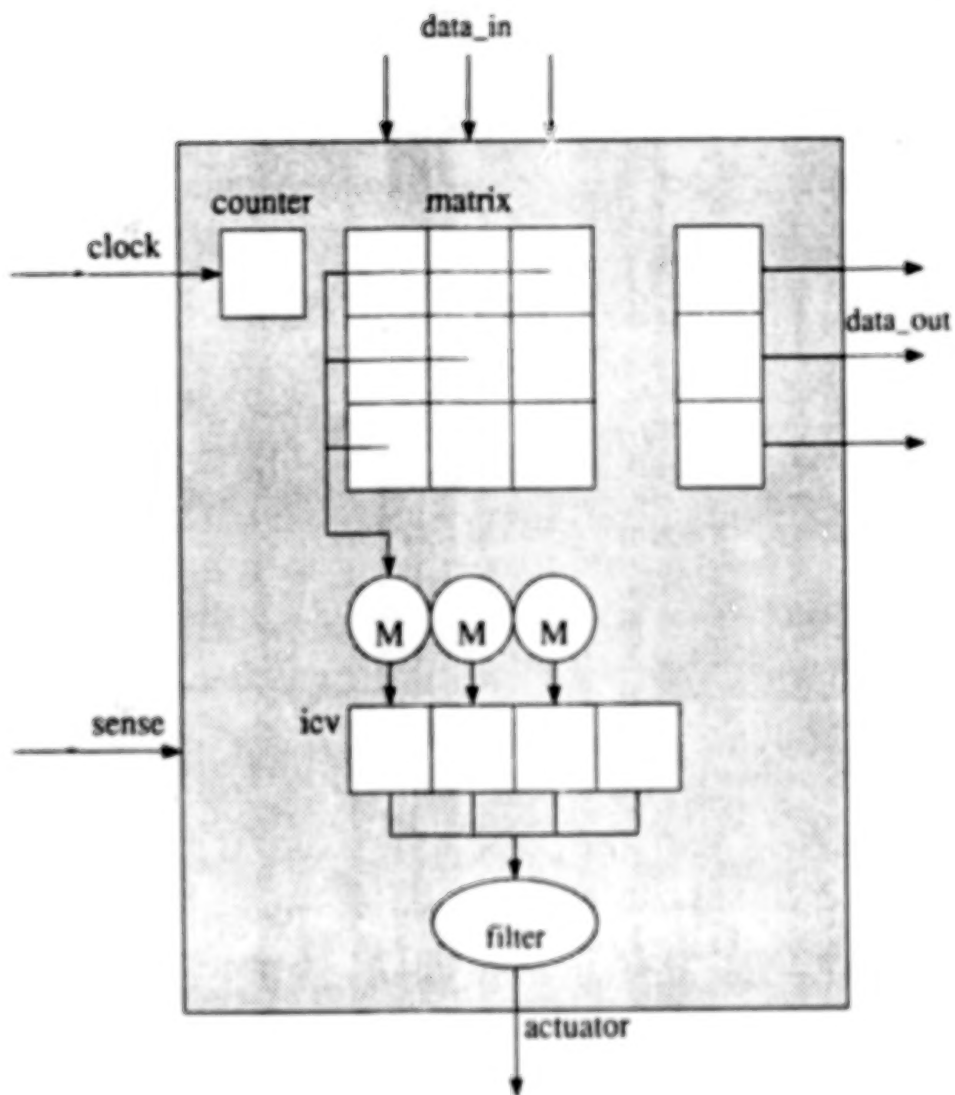
At each step, all non-faulty processes agree on their output iff the total number of processors exceeds three times the number of faulty processors.

## The Device Design

Goal: Design 4 identical circuits which, when operating synchronously, achieve Byzantine agreement.



## A Process Internal State



## Process Steps

0: data\_out[i]  $\leftarrow$  sense,  $i \in \{0,1,2\}$   
icv[3]  $\leftarrow$  sense  
clock  $\leftarrow$  clock+1

1: m[0,i]  $\leftarrow$  input[i],  $i \in \{0,1,2\}$   
data\_out[0]  $\leftarrow$  input[1]  
data\_out[1]  $\leftarrow$  input[0]  
data\_out[2]  $\leftarrow$  input[0]  
clock  $\leftarrow$  clock+1

2: m[1,i]  $\leftarrow$  input[i],  $i \in \{0,1,2\}$   
data\_out[0]  $\leftarrow$  m[0,2]  
data\_out[1]  $\leftarrow$  m[0,2]  
data\_out[2]  $\leftarrow$  m[0,1]  
clock  $\leftarrow$  clock+1

3: m[2,i]  $\leftarrow$  input[i],  $i \in \{0,1,2\}$   
clock  $\leftarrow$  clock+1

4: icv[0]  $\leftarrow$  majority(m[0,0], m[1,2], m[2,1])  
icv[1]  $\leftarrow$  majority(m[0,1], m[1,0], m[2,2])  
icv[2]  $\leftarrow$  majority(m[0,2], m[1,1], m[2,0])  
clock  $\leftarrow$  clock+1

5: Actuator  $\leftarrow$  filter(icv)  
clock  $\leftarrow$  clock+1

6: clock  $\leftarrow$  clock+1

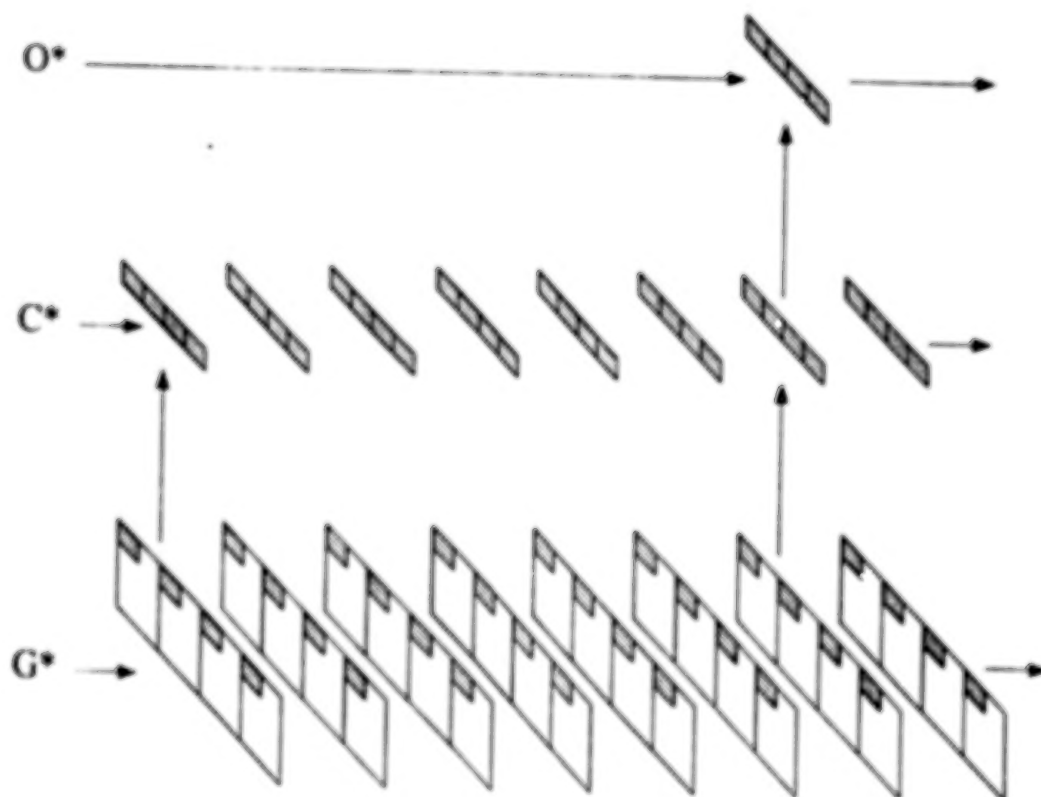
7: clock  $\leftarrow$  clock+1

## Summary of Device Design

1. Four identical devices.
2. Only internal and external data flow specified, data width not.
3. Filter function constrained to tolerate ICV rotations.

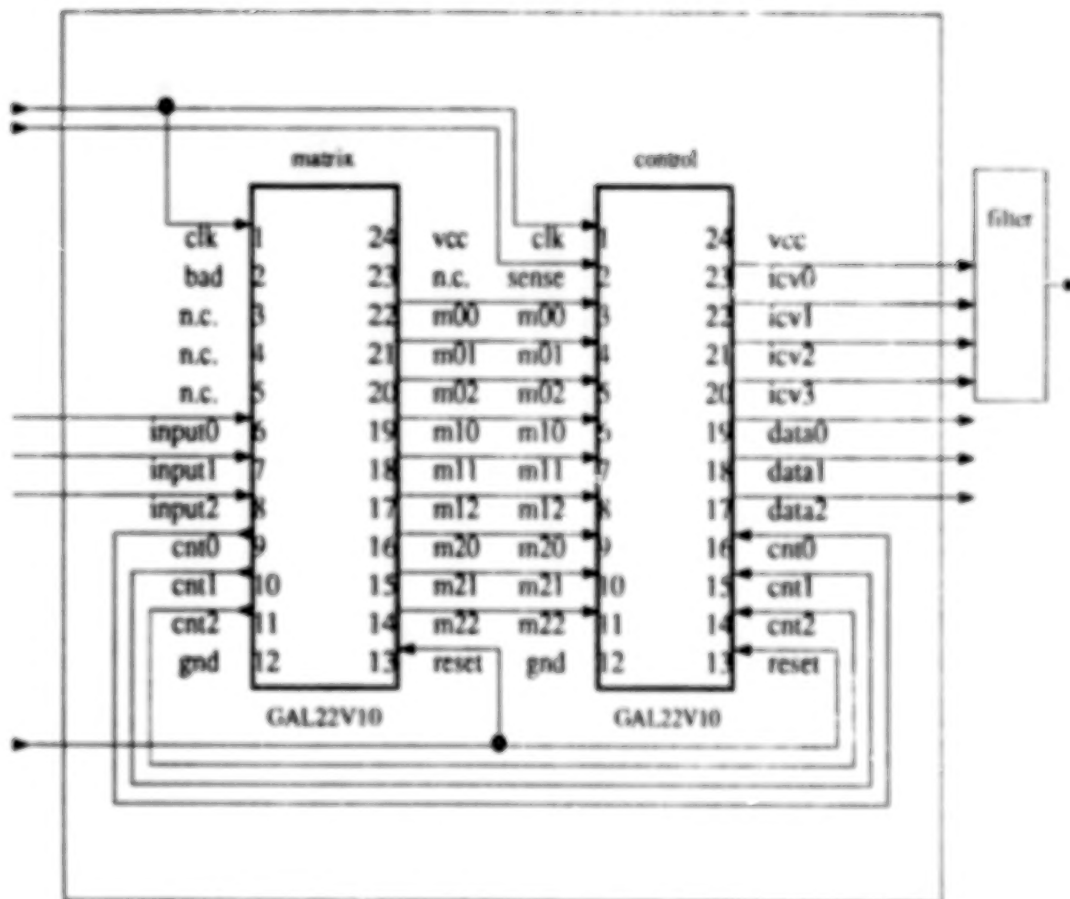


## Correctness of Device Design



# Device Implementation

by Larry Smith



295  
511-61

N91-17570

Verifying an Interactive Consistency Circuit:  
A Case Study in the Reuse of a Verification Technology

319692 ✓  
P.31  
Mark Bickford  
Mandayam Srivas

Odyssey Research Associates, Inc.  
301A Harris B. Dates Drive  
Ithaca, NY 14850.

This talk presented the work done at ORA for NASA-LRC in the design and formal verification of a hardware implementation of a scheme for attaining interactive consistency (byzantine agreement) among four microprocessors. The microprocessors used in the design are an updated version of a formally verified 32-bit, instruction-pipelined, RISC processor, MiniCayuga. The 4-processor system, which is designed under the assumption that the clocks of all the processors are synchronized, provides "software control" over the interactive consistency operation. Interactive consistency computation is supported as an explicit instruction on each of the microprocessors. An identical user program executing on each of the processors decides when and on what data interactive consistency must be performed.

This exercise also served as a case study to investigate the effectiveness of reusing the technology which had been developed during the MiniCayuga effort for verifying synchronous hardware designs. MiniCayuga was verified using the verification system Clio which was also developed at ORA. To assist in reusing this technology a computer-aided specification and verification tool was developed. This tool specializes Clio to synchronous hardware designs and significantly reduces the tedium involved in verifying such designs. The talk presented the tool and described how it was used to specify and verify the interactive consistency circuit.

## Summary

### Achievements

1. Formalization of abstract Byzantine agreement algorithm.
2. Use of this algorithm to specify a hardware device.
3. A mechanically checked proof that the device design is correct.
4. The implementation of the device from the low-level design.

### Limitations

1. Assumes synchronized behavior of the processes.

**Verifying an Interactive Consistency  
Circuit:**

*A Case Study in the Reuse of  
a Verification Technology*

Mark Bickford  
Mandayam Srivas

Odyssey Research Associates, Inc.  
301A Harris B. Dates Drive  
Ithaca, NY 14850.

## Objectives of the Work

- Design an efficient hardware implementation for a 4-processor architecture
- Use verified MiniCayuga's in the design
- Verify the design
- Reuse MiniCayuga verification technology
  - A method of modeling synchronous hardware designs in the Clio verification system
  - Formalizing a class of properties most commonly encountered in verifying designs
  - A "standard" proof strategy

# Clio: A functional Language Based Verification System

- Caliban: A modern functional language  
eg., higher order functions, data types,  
lazy, etc.

$$\text{least } P \ x = \ x, \ P \ x$$
$$\text{least } P \ x + 1$$

- Assertion Level: Full FOPC with equality  
on Caliban terms

$$\text{Prop} := (P)(x) \rightsquigarrow [\neg \text{'!(least } P \ x)\text{'} = \text{'True'}]$$
$$\vee \text{' } P \ (\text{least } P \ x) \text{' = 'True'}$$

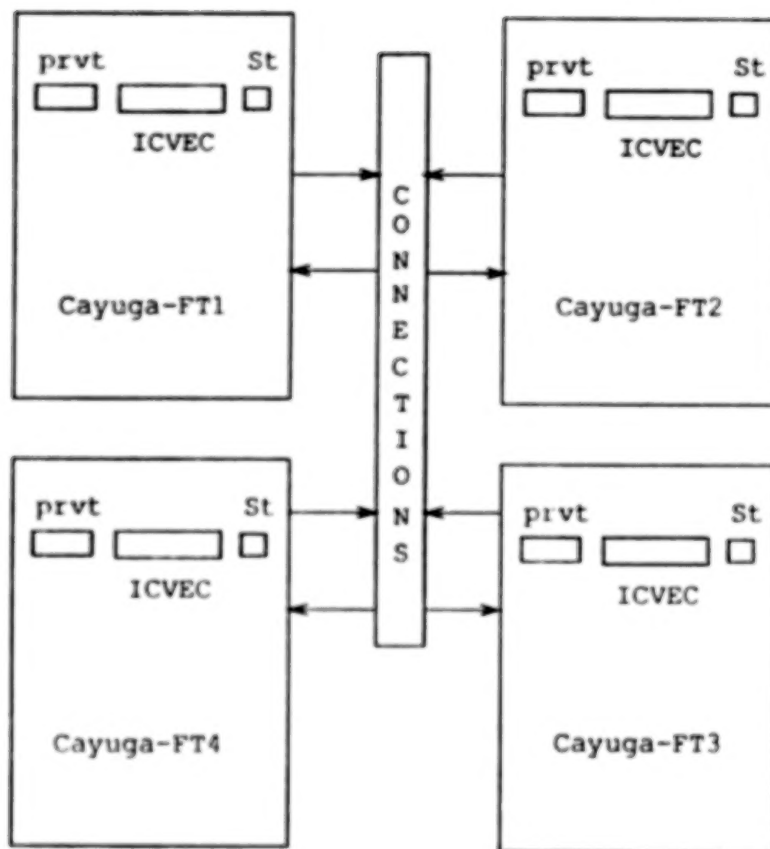
- Interactive Theorem Prover
  - rewriting
  - Induction
    - structural
    - Fixed point
  - Other FOPC proof strategies

## **Presentation Outline**

- IC circuit design
- The computer-aided hardware verification tool
- How we verified it
- General observations about the effort



## The Hardware Design: Overview



Two new instructions:

ICOP REG            - initiates and co-orinates  
                     IC computation  
MOVE SREG REG - moves special REG to  
                     general REG

|| check if voter is free

Notfree        MOVE STATUS REG1  
                     JIF REG1 Notfree  
                     ICOP REG2

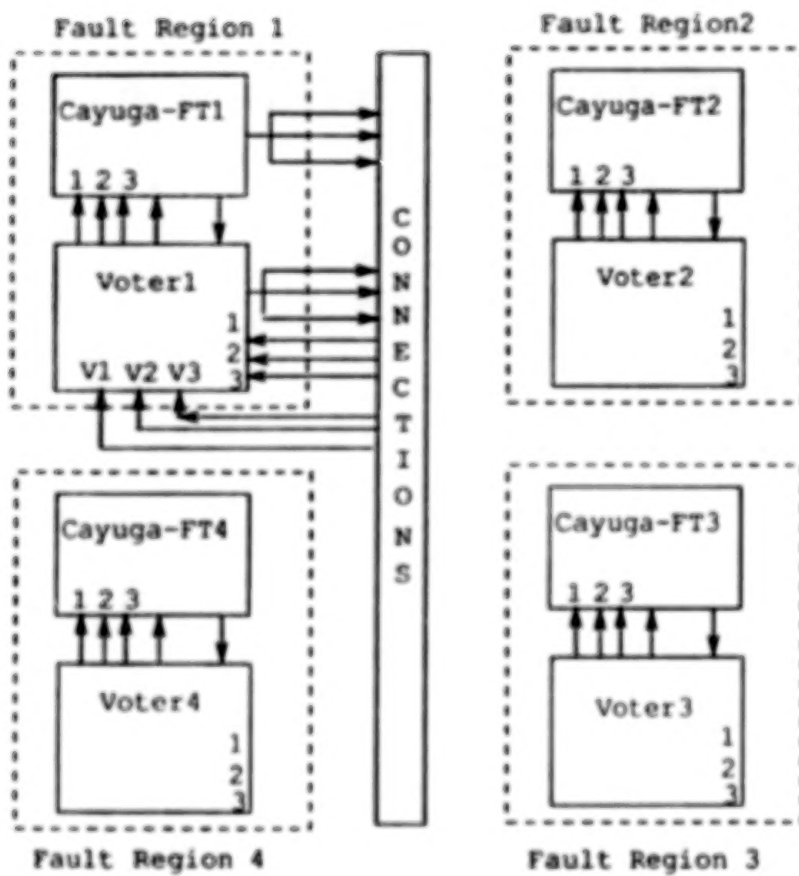
|| check if IC computation is complete

Notready       MOVE STATUS REG1  
                     JIF REG1 Notready

|| move the results of IC to general registers

                     MOVE SREG0 REG3  
                     MOVE SREG1 REG4  
                     MOVE SREG2 REG5

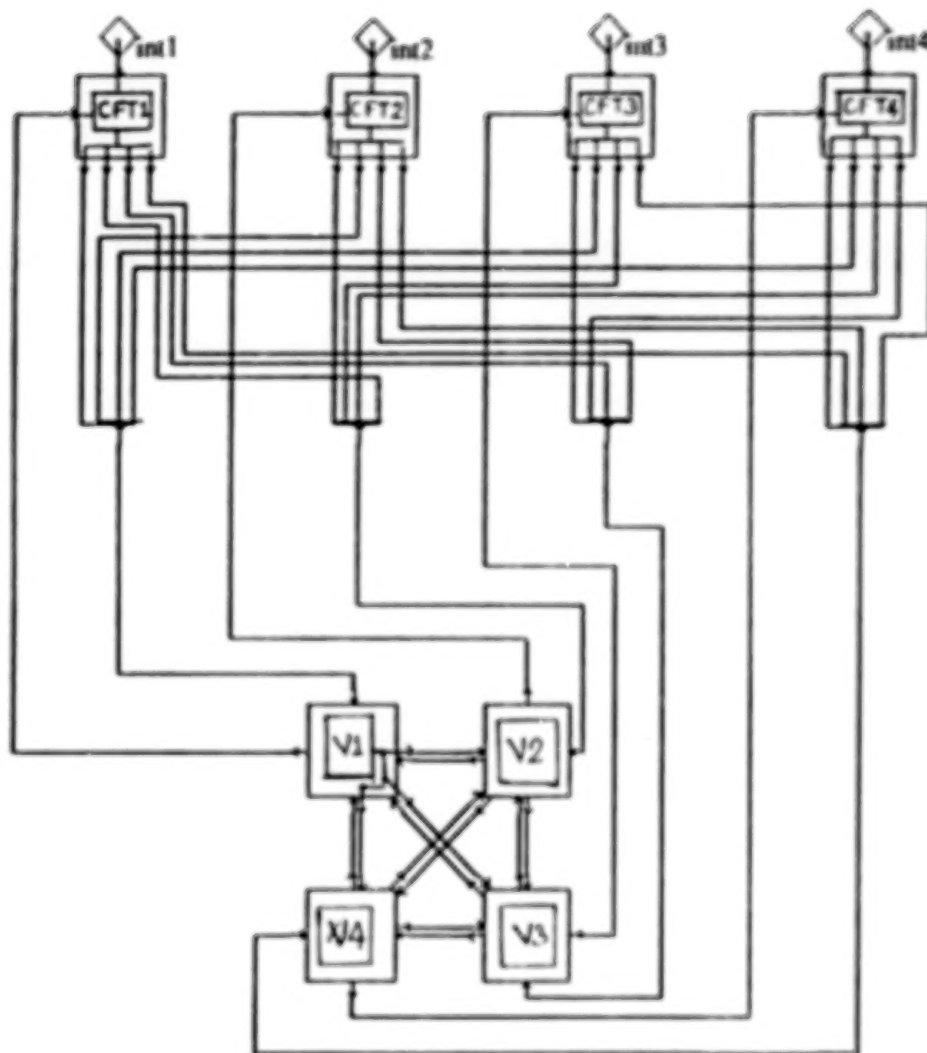
## The Hardware Design: Overview



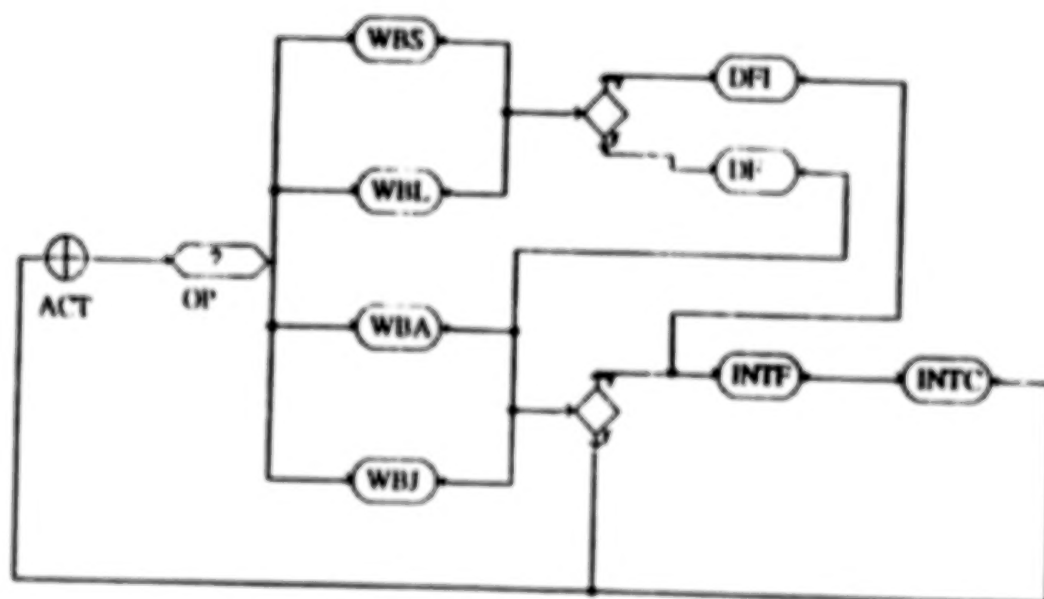
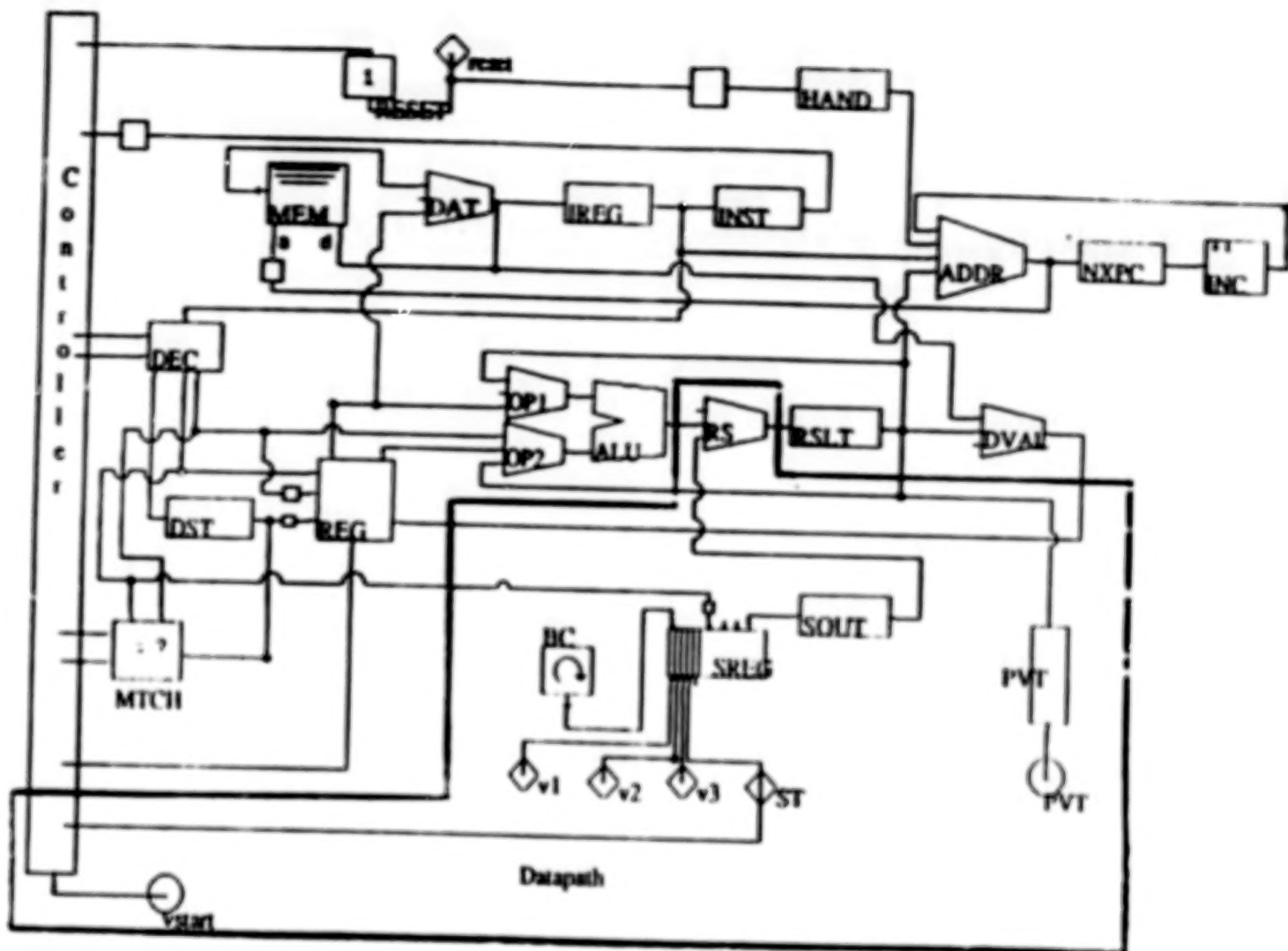
- voter separate from processor: modularity
- point-to-point connection: electrical isolation
- serialize data transfers: number of pins  
Vs. time
- Fault region: processor, voter, and the connections they feed

- no absolute indexing scheme for processors/voters
  - relative indexing scheme:  $succ$ ,  $succ^2$ ,  $succ^3$
  - IC vectors will be stored in the processors in the order of their successors
- Underlying assumption: clocks are synchronized with at most a bounded skew
  - hold sender's signal stable for one phase longer than needed

## IC System Design Behavior



- *Initiate*: draw the attention of voter (1)
- *Load*: transfer private values (2)
- *Exchange*: exchange received values (6)
- *Compute*: compute and store IC vector (3)

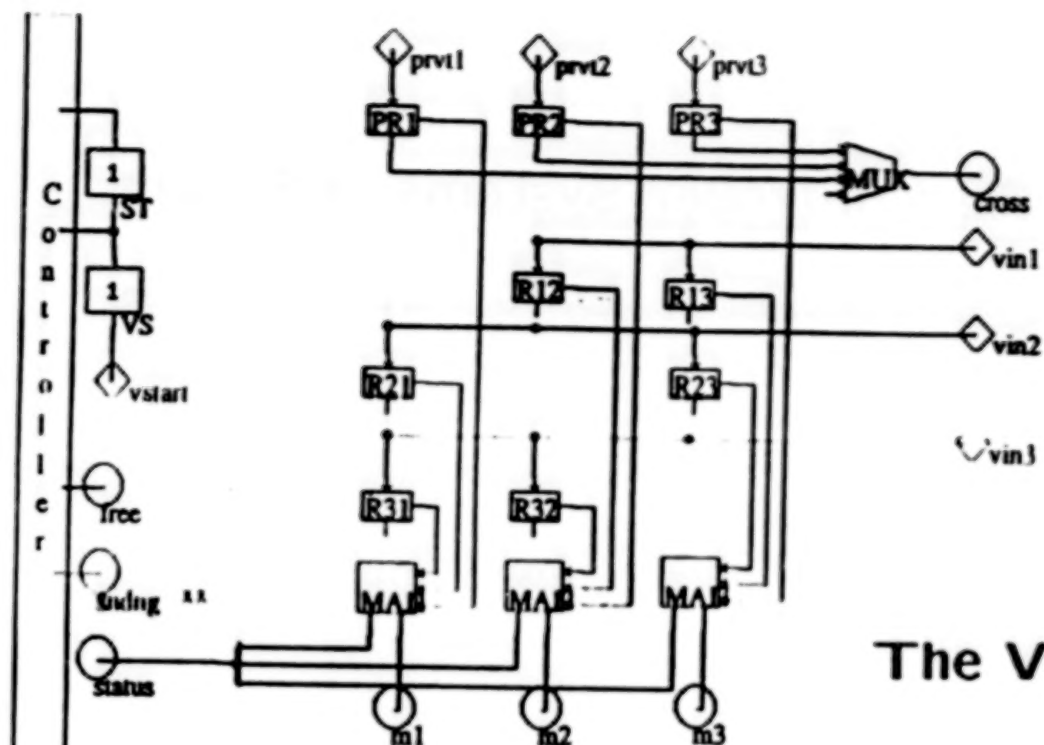


9  
Controller State Machine

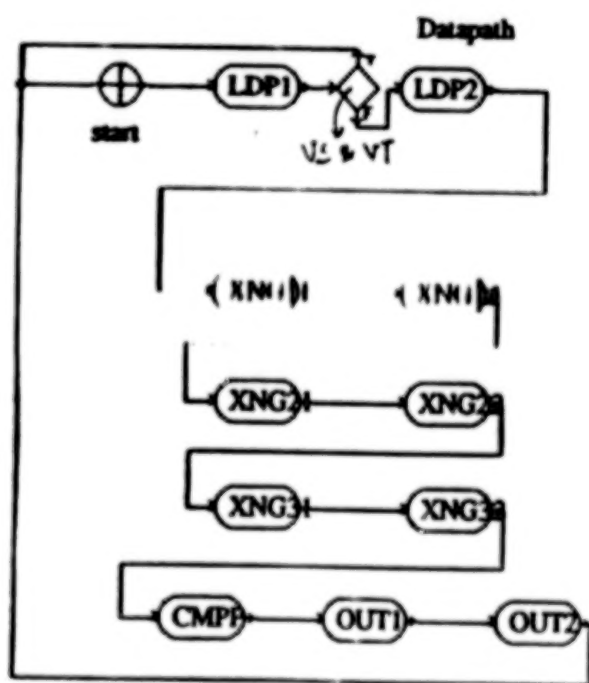
## **MiniCayuga Processor: Summary**

- Inspired by Cayuga (Cornell University)
- 32-bit RISC processor
- Design characteristics
  - 32 general purpose registers
  - small and simple instruction set
  - 3-stage instruction pipeline: fetch, compute, writeback
  - delayed jump, pipeline stalling, internal forwarding
  - interrupt





The Voter Circuit



## What do we prove ?

### Assuming

- every Cayuga-FT is about to execute an ICOP,
- every Voter is ready to vote, and
- there is at most one faulty region,

then, 12 cycles later the system state will satisfy the following conditions:

- The IC vectors in the processors are identical "up to rotation."
- The IC vectors are correct w.r.t. to the processor private values 12 cycles earlier.

## **A Computer-Aided Verification Tool**

- Specializes Clio to the domain of *finite state controller systems*
- Design specification generation
- Verification condition formulation
- Automatic proof support

## Finite State Controller Systems (FSCS)

- Central Controller + Data Path components
- Component behavior is specified as a set of *actions*
- Controller is specified as an FSM which schedules a set of *actions* on the components.
- Timing Model
  - Every transition corresponds to a clock cycle (with multiple phases)
  - An action may have zero or more units (phases) of delay
  - Actions are synchronized with state transitions

## Specification technology reused

- a method of formalizing the intended operational model of an FSCS in Caliban/Clio

`designspecgen ::`

`data-path-structure ->`

`controller-structure ->`

`controller-schedule ->`

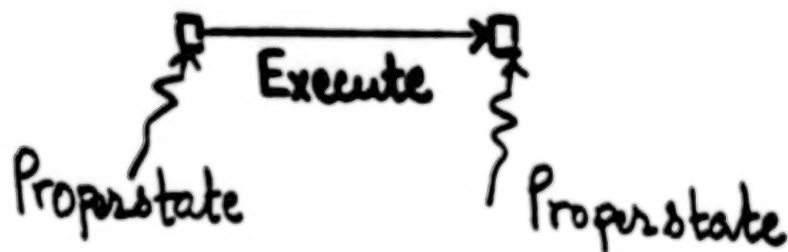
`actions-behavior -> design-spec`

`Execute :: STATE -> STATE`

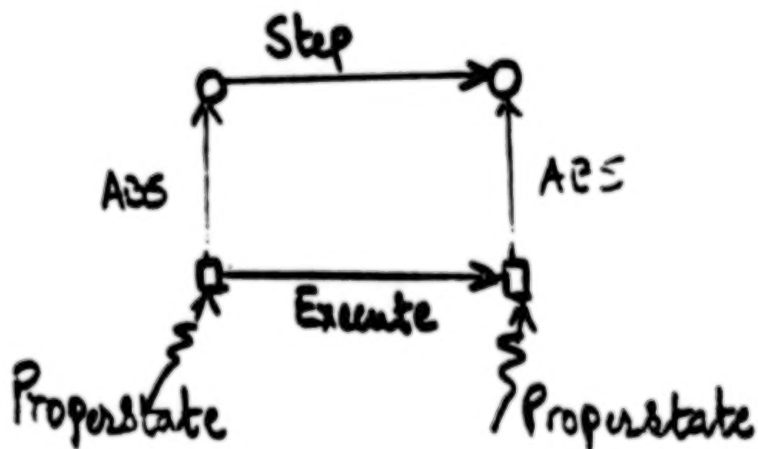
"single clock cycle behavior of design"

## Proof technology shared

- Form of the most commonly proved conditions
  - Invariant conditions

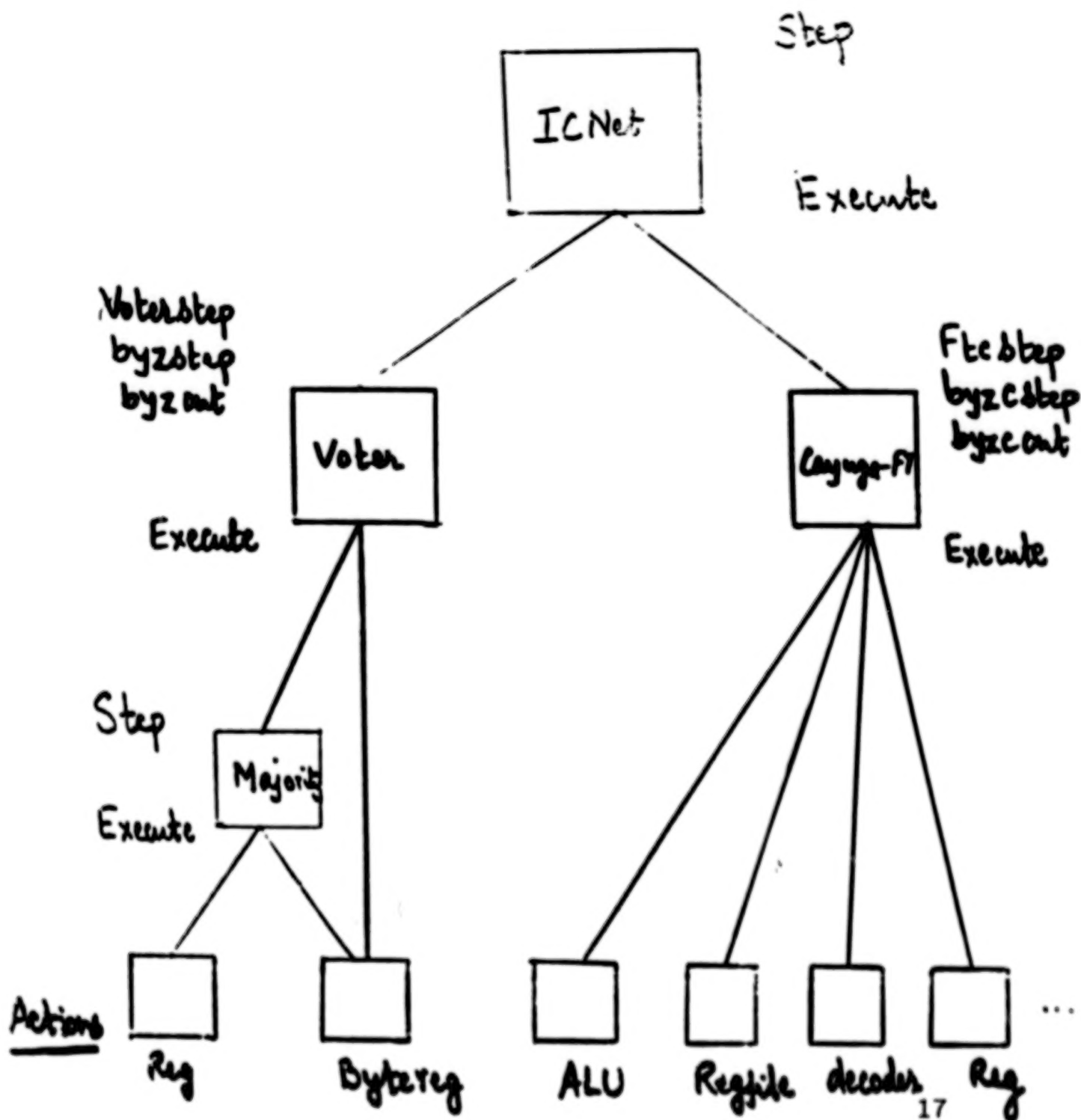


- Advance conditions



- Proof strategy: "controlled symbolic evaluation (rewriting) with selective case-splits"

# The Specification Hierarchy



## Rationale for the hierarchy

- Decompose proofs into manageable units
- Need for the black level
  - introduce "error" actions
  - type of Execute is different from that of action
- Implication of intermediate levels
  - *pro*: proof can take "bigger" steps
  - *con*: must come up with intermediate abstract specification



## Top Level Specification

```
||IcNetState ::~ <<(INDEX -> FTCstate),  
||              (INDEX -> Voterstate), Interrupts>>  
  
IcNetStep <<ftc,vtr, int:rest>> =  
  <<newftc,newvtr ,rest>>  
  where newftc index  
    = fault_ftc_step index ftc (ftcinput index)  
  newvtr index  
    = fault_vtr_step index vtr (vtrinput index)  
  ftcinput index  
    = make_ftc_in (select_int index int)  
                  (fault_to_proc index ftc vtr)  
  vtrinput index  
    = Voterinput index ftc vtr  
                                     (ftcinput index )  
  
fault_ftc_step index s in =  
  FtCayugaStep (s index) in , ~(faulty index)  
  byzCayugaStep (s index) in  
  
fault_vtr_step index s =  
  voterstep (s index) , ~(faulty index)  
  byzstep (s index)
```

...

## Formal Statement of Correctness

MainTheorem :=

Preconditions 's' => ResultConsistent 's'

ResultConsistent 's' :=

Consistent 'icvec s (Iterate #12 IcNetStep s)'

Consistent 'array' :=

'faulty index'='False' =>

IndexConsistent 'array' 'index'

IndexConsistent 'array' 'index' :=

('faulty (succ index)'='False'=>

'(array index).succ'='array (succ index)')

& ('faulty (succ2 index)'='False'=>

'(array index).succ2'='array (succ2 index)')

& ('faulty (succ3 index)'='False'=>

'(array index).succ3'='array (succ3 index)')

Preconditions 's' :=

Proper\_icnet 's' & Sync 'LDP1' 's' & All\_go 's'

Sync 'cs' '<<ftc,vtr,inlist>>' :=

('faulty ONE' = 'False' =>

'control (vtr ONE)'='cs')

& ('faulty TWO' = 'False' =>

'control (vtr TWO)'='cs')

& ('faulty THREE' = 'False' =>

'control (vtr THREE)'='cs')

& ('faulty FOUR' = 'False' =>

'control (vtr FOUR)'='cs')

All\_go 's' :=

('faulty ONE'='False' =>

('go\_of (vtr s ONE)'='False' & 'go\_signal s ONE'='GO

& ('faulty TWO'='False' =>

('go\_of (vtr s TWO)'='False' & 'go\_signal s TWO'='GO

& ('faulty THREE'='False' =>

('go\_of (vtr s THREE)'='False' & 'go\_signal s THREE'

& ('faulty FOUR'='False' =>

('go\_of (vtr s FOUR)'='False' & 'go\_signal s FOUR'='

## **The proof strategy reused**

*"controlled symbolic execution of design"*

1. Instantiate the states of components and inputs with appropriate symbolic constants.
2. Add all the conditions on the constants implied by the preconditions of the theorem as hypothesis.
3. Symbolically evaluate design.
4. Try *case-splitting* on all the conditionals automatically.
5. If either of the previous two steps seem to take too long, then case-split on the controller states and inputs before symbolic evaluation (step 3).

## **New technology needed**

- Modeling faulty behavior
- Specification
  - determining the right hierarchy
  - writing intermediate "abstract" spec
  - defining abstraction function (ABS)
- Proof: "design level properness" implies "abstract level properness"

## General Observations

- An engineering-oriented verification experience  
Lilith → MiniCayuga → IC circuit
- Methodology: top-down + bottom-up
- Level of effort: 1 man year
  - building the tool
  - developing designs
  - verification

## **Verification Effort Milestones**

- formulated a top level correctness statement
- designed and verified a simple voter circuit
- specified voter and processor for a continuous voting scheme
- designed and verified second voter design

- discovered continuous voting scheme was "hard to synchronize"
- respecified voter and processor for a voting-on-demand scheme
- redesign and reverify voter
- verified overall system
- verified processor



- To integrate theorem proving based verification technology into the design process we need:
  - more machine assistance
  - domain specialization
  
- The next step ?
  - A useful way of reporting failed proof attempts
  - Interaction with motivated and patient engineering design teams and projects

326  
512-61  
319693  
P-28  
N91-17571

**Hardware Verification  
at  
Computational Logic, Inc.**

**Bishop C. Brock  
Warren A. Hunt, Jr.**

Computational Logic Incorporated  
1717 West Sixth Street, Suite 290  
Austin, Texas 78703-4776

+1 512 322 9951  
Brock@CLI.COM, Hunt@CLI.COM

15251-107

## **Talk Topics**

- **Hardware Verification: What Is It?**
- **Formal Methods: What Good Are They?**
- **Verification Methodology**
- **Present Accomplishments**
- **Expected Near Term Results**
- **Present Trends**
- **Future Directions**
- **Collaborations and Technology Transfer**
- **Technology Enablers**
- **Conclusions**

## **Hardware Verification: What Is It?**

The mathematical formalization of the specification of any (all) aspects of hardware design.

We specifically are interested in the design of hardware for digital computing.

### **Goals:**

- Completely replace programmer's manuals, timing diagrams, interface specifications, power requirements, etc. with clear precise formulas.
- Provide a perfectly clear foundation upon which systems can be built.

## **Formal Methods: What Good Are They?**

Formal methods in the U.S. have a bad credit rating.

Over the years, good mechanized software verification systems have been constructed.

Good software verification tools are being extended to include hardware verification, thus providing good systems verification tools.

Hardware verification seems more tractable than software verification:

- few, repeatedly-used, low-level constructs;
- specification domain is less abstract (fairly concrete); and
- formal methods can be used incrementally.

Last point is critical, note Bryant's work.

## **Our Verification Methodology**

We employ the Boyer-Moore logic to:

- write design specifications;
- write behavioral specifications; and
- record relations.

The Boyer-Moore theorem prover

- insures that definitions are well formed;
- checks that proofs are correct; and
- manages our evolving database of facts.

## **Present Accomplishments**

Our application of formal methods to hardware specification and verification include:

- Core RISC specification;
- FM8502 microprocessor verification;
- verification of circuits using standard TTL components;
- a formalization of a simple HDL; and
- verified synthesis of combinational circuits.

Let us consider several in more detail.

## **Core RISC**

Bill Bevier has formally specified a set of instructions that characterize a Core RISC-compliant processor. This formalization includes:

- byte, half-word, and long-word memory accesses;
- Boolean, natural number, and integer ALU operations;
- a minimum register set; and
- an exception mechanism.

The emphasis here has been on mathematically modeling the instruction set.

Our study of RISC architectures indicates that we need to be able to model multi-phase clocking schemes before we attempt to design a build a verified Core RISC processor. This effort is ongoing.



## **The FM8502 Fabrication**

Currently, our primary effort involves the fabrication of the FM8502 microprocessor.

This fabrication effort is a test-of-concept; that is, can we manufacture formally modeled circuits and get them working?

The FM8502 microprocessor is a 32-bit general purpose microprocessor with:

- 32-bit addressing;
- 16 general-purpose registers;
- two-address architecture;
- 5 addressing modes;
- a 16-function ALU
- extensive flag support; and
- little else.



# MODE OPERAND DESCRIPTION

00	Rn	Register Direct
01	(Rn)	Register Indirect
10	-(Rn)	Register Indirect Pre-decrement
11	(Rn)+	Register Indirect Post-increment

OP-CODE	OPERATION	DESCRIPTION	STORE-CC	CONDITION
0000	b ← a	Move	0000	Carry clear
0001	b ← a + 1	Increment	0001	Carry set
0010	b ← a + b + c	Add with carry	0010	Overflow clear
0011	b ← b + a	Add	0011	Overflow set
0100	b ← 0 - a	Negation	0100	Not negative
0101	b ← a - 1	Decrement	0101	Negative
0110	b ← b - a - c	Subtract with borrow	0110	Not zero
0111	b ← b - a	Subtract	0111	Zero
1000	b ← a >> 1	Rotate right through carry	1000	Higher
1001	b ← a >> 1	Arithmetic shift right	1001	Lower or same
1010	b ← a >> 1	Logical shift right	1010	Greater or equal
1011	b ← b XOR a	XOR	1011	Less
1100	b ← b OR a	OR	1100	Greater
1101	b ← b AND a	AND	1101	Less or equal
1110	b ← NOT a	NOT	1110	True
1111	b ← a	Move	1111	False

## **The FM8502 Implementation Specification**

To be able to manufacture the FM8502 with some precision, we have been working on the formalization of an HDL.

We will prove the correctness of our HDL description of the FM8502, and then translate our HDL description into a commercial HDL.

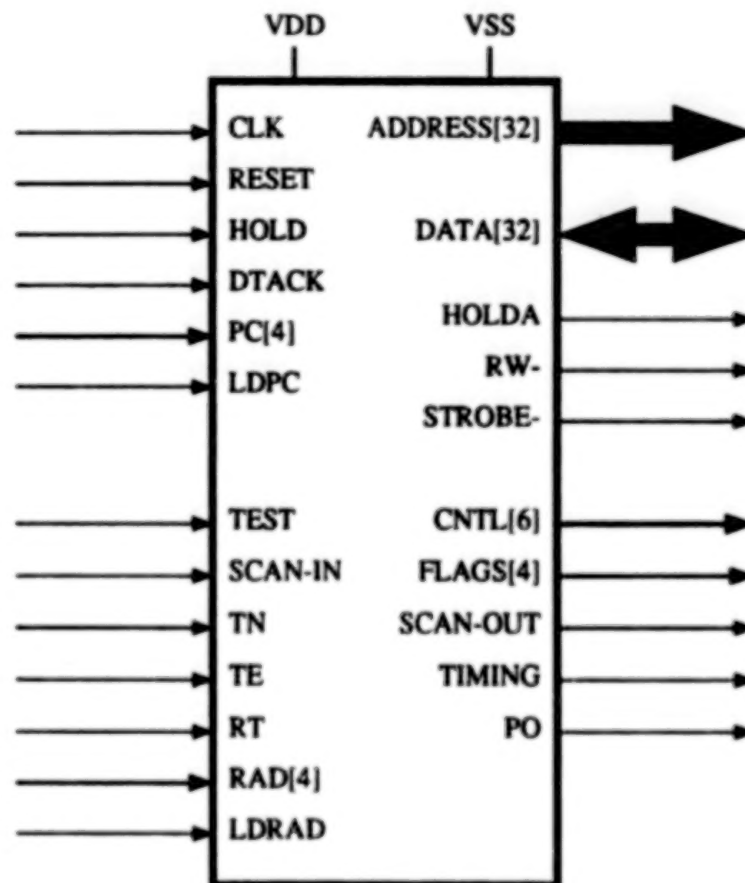
Our HDL provides our lowest-level model for the FM8502 implementation:

- every internal gate and register is described;
- every I/O pad is defined; and
- we expect to validate our test vectors directly on our HDL description.

Our HDL specification also includes all of the internal test logic.

## The FM8502 Pinout

Below is a pictorial diagram of the FM8502 pinout. Quite a number of pins are allocated to testing purposes.



## **A Formal HDL**

Our HDL is structured like commercial HDL's:

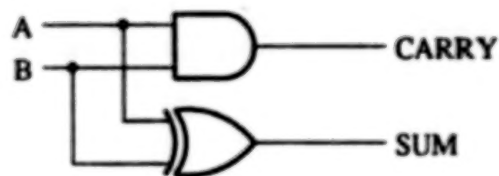
- netlist based;
- heirarchicaly structured;
- occurence-oriented; and
- allows multiple views of circuits.

We have a formal specification of our HDL:

- a predicate recognizes well-formed circuits; and
- several interpreters define the semantics.

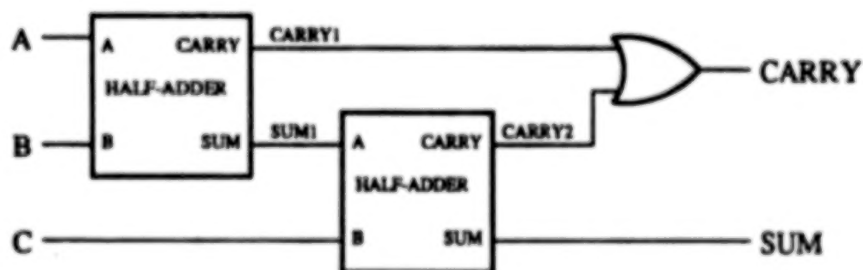
## HDL Examples of Circuits

```
' (HALF-ADDER (A B)
  (SUM CARRY)
  (( G0 (SUM)   B-XOR(A B) )
    ( G1 (CARRY) B-AND(A B) )))
```



The following full-adder specification refers twice to the half-adder specification above.

```
' (FULL-ADDER (A B C)
  (SUM CARRY)
  (( T0 (SUM1 CARRY1) HALF-ADDER(A B) )
    ( T1 (SUM CARRY2)  HALF-ADDER(SUM1 C) )
    ( T2 (CARRY)       B-OR(CARRY1 CARRY2) )))
```



## **Verified Synthesis**

We perform synthesis by

- writing circuit generator programs;
- verifying the circuit generator programs; and
- then running the generators to produce provably correct circuits.

In other words, after a circuit has been generated we need not inspect it for the Boolean correctness.

## **An ALU Generator**

We have an arbitrary size, 16-function ALU generator which is:

- programmable -- ALUs with different internal structure can be produced;
- "intelligent" -- internal buffers are only added when needed; and
- has been verified to generate correct  $n$ -bit, gate-level ALU descriptions.

Simple translators can convert the ALU descriptions into conventional CAD languages (e.g., VHDL).

To replay the proof only takes about 20 (Sun 3) minutes.



## ALU Generator Output Summary

Summarized below are some characteristics of the ALUs generated by our verified ALU generator.

ALU Characteristics			
Size	Gate Count	Fanout	Delay
1 bit	126	8	12
2 bits	149	8	14
4 bits	196	8	17
8 bits	297	8	22
16 bits	491	8	26
32 bits	880	8	30
64 bits	1665	8	35
128 bits	3227	8	39

**Payoff:** It only takes 0.6 seconds to generate a correct 32-bit ALU, 1.3 seconds for a 64-bit ALU, and 3.1 seconds for a 128-bit ALU.

## **Expected Near Term Results**

Several projects underway which will conclude this year are:

- an ability to verify sequential circuits generators;  
and
- the fabrication of the FM8502 microprocessor.

We are using both combinational and sequential logic synthesis techniques in the fabrication of the FM8502.

We will be able to generate a correct  $n$ -bit microprocessor (so long as the word size is large enough to contain FM8502 instructions.)

We will generate a gate-array specification directly.

We are generating our test-vectors directly from our formal circuit specifications.

## **Present Trends**

There is increasing interest in:

- boolean comparison -- which should lead the way to more general purpose techniques;
- register-transfer specifications with circuit verification;
- formalization of self-timed circuits;
- formalization of timing behavior; and
- transformational systems.

These trends are all indicative of increased use of formal techniques for hardware specification and verification.

And these techniques are being applied incrementally.

## **Future Directions**

In the future we hope to:

- formalize a subset of VHDL (using our Ada formalization experience);
- perform tool verification (e.g., logic minimizer, tautology checkers);
- verify a Core RISC microprocessor with memory management; and
- continue our work on formalizing hardware interfacing and use specifications.

This last item is hardest and has the biggest payoff.

## **Industrial Collaborations**

We have been working with DEC for two years.

Motorola may attempt the specification (and possibly the verification) of one of their microcontrollers.

## **Technology Transfer**

We highly value interactions with industry; we all profit.

Our formal techniques may be used incrementally, i.e., "creeping formalization."

Industry first employs our techniques for (unambiguous) specification, later for verification.

Specification is a big problem for industry -- formal specification allows analysis without exhaustive testing.

## **Technology Enablers**

Is the state-of-the-art separating further from the state-of-the-practice?

To enable the use of formal techniques in hardware design we need to.

- train more engineers with formal methods (not train mathematicians to be engineers);
- make existing tools and techniques more accessible to engineers; and
- make formal techniques the most economical method of hardware validation.

A big success or two would help us get industry's attention.

## **Conclusions**

Formal methods can be used to provide accurate specifications.

Hardware verification provides increased assurance of circuit correctness.

Formal techniques provide a good growth path; they scale up well.

The credit rating of formal techniques is improving.

### **Goals:**

- Completely replace programmer's manuals, timing diagrams, interface specifications, power requirements, etc. with clear precise formulas.
- Provide a perfectly clear foundation upon which systems can be built.

347

5/3 -60

319694

N91-17572

233

# Generic Interpreters and Microprocessor Verification

Phillip J. Windley

*Department of Computer Science  
University of Idaho*

August, 1990

This work was sponsored under Boeing Contract NAS1-18586, Task Assignment No. 3,  
with NASA-Langley Research Center.



104-11235

## *Outline*

- Introduction
- Generic interpreters
- Microprocessor Verification
- Future Work

## *Microprocessor Verification*

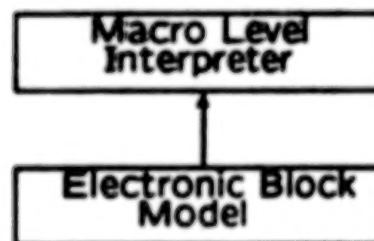
- VIPER, the first commercially available, "verified" microprocessor, has never been formally verified.
- The proof was not completed even though 2 years were spent on the verification.

## ***Microprocessor Verification*** (continued)

- Our research is aimed at making the verification of large microprocessors tractable.
- *Our objective is to provide a framework in which a masters-level student can verify VIPER in 6 person-months.*

## *Determining Correctness*

In VIPER (and most other microprocessors), the correctness theorem was shown by proving that the electronic block model implies the macro-level specification.



## *The Problem*

(continued)

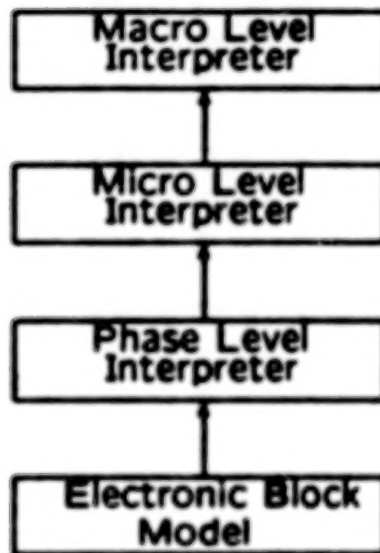
- Microprocessor verification is done through case analysis on the instructions in the macro level.
- The goal is to show that when the conditions for an instruction's selection are right, the electronic block model implies that it operates correctly.
- A lemma that the EBM correctly implements each instruction can be used to prove the top-level correctness result.

## ***The Problem***

Unfortunately, the one-step method doesn't scale well because

- The number of cases gets large.
- The description of the electronic block model is very large.

## *Hierarchical Decomposition*



- A microprocessor specification can be decomposed hierarchically.
- The abstract levels are represented explicitly.

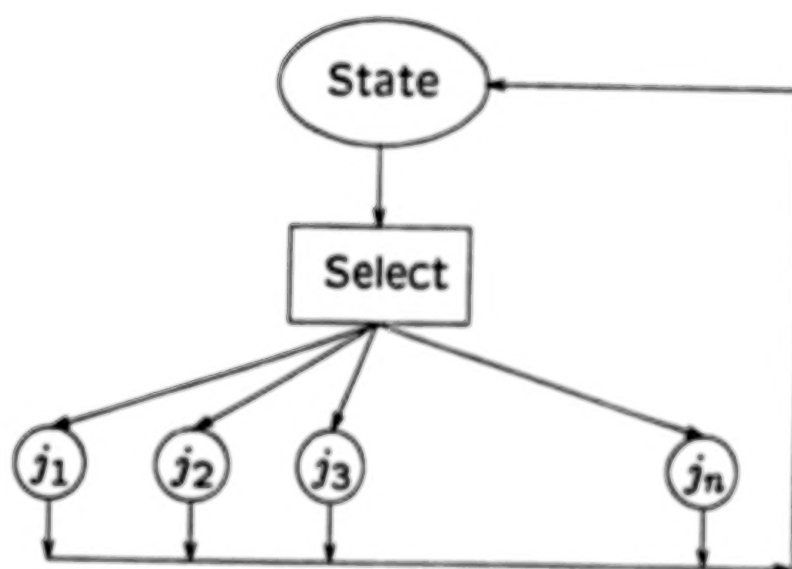
## *Interpreters*

An abstract model of the different layers in the hierarchy provides a methodological approach to microprocessor verification.

- The model drives the specification.
- The model drives the verification.



## *Interpreters* (top level)



## *Specifying an Interpreter* (overview)

We specify an interpreter by:

- Choosing a  $n$ -tuple to represent the state, **S**.
- Defining a set of functions denoting individual interpreter instructions, **J**.
- Defining a next state function, **N**.
- Defining a predicate denoting the behavior of the interpreter, **I**.

## *Verifying an Interpreter* (overview)

We verify an interpreter, **I** with respect to its implementation **M** by showing

$$\mathbf{M} \Rightarrow \mathbf{I}.$$

To do this, we will show that every instruction in **J** can be correctly implemented by **M**:

$$\forall j \in \mathbf{J}.$$

$$\mathbf{M} \Rightarrow (\forall t: \text{time}.$$

$$C(t) \Rightarrow s(t + n) = j(s(t)))$$

where  $C$  represents the conditions for instruction  $j$ 's selection.

## ***AVM-1***

We have designed and are verifying a micro-computer with interrupts, supervisory modes and support for asynchronous memory.

- The datapath is loosely based on the AMD 2903 bit-sliced datapath.
- The instruction format is very simple.
- The control unit is microprogrammed.

## *AVM-1's Instruction Set* (subset)

Opcode	Mnemonic	Operation
000000	JMP	jump on 16 conditions
000001	CALL	call subroutine
000010	INT	user interrupt
000110	LD	load
000111	ST	store
010000	ADD	add (3-operands)
011011	SUBI	subtract immediate (2-operands)
011111	NOOP	no operation

- The architecture is load-store.
- The instruction set is RISC-like.
- There is a large register file.

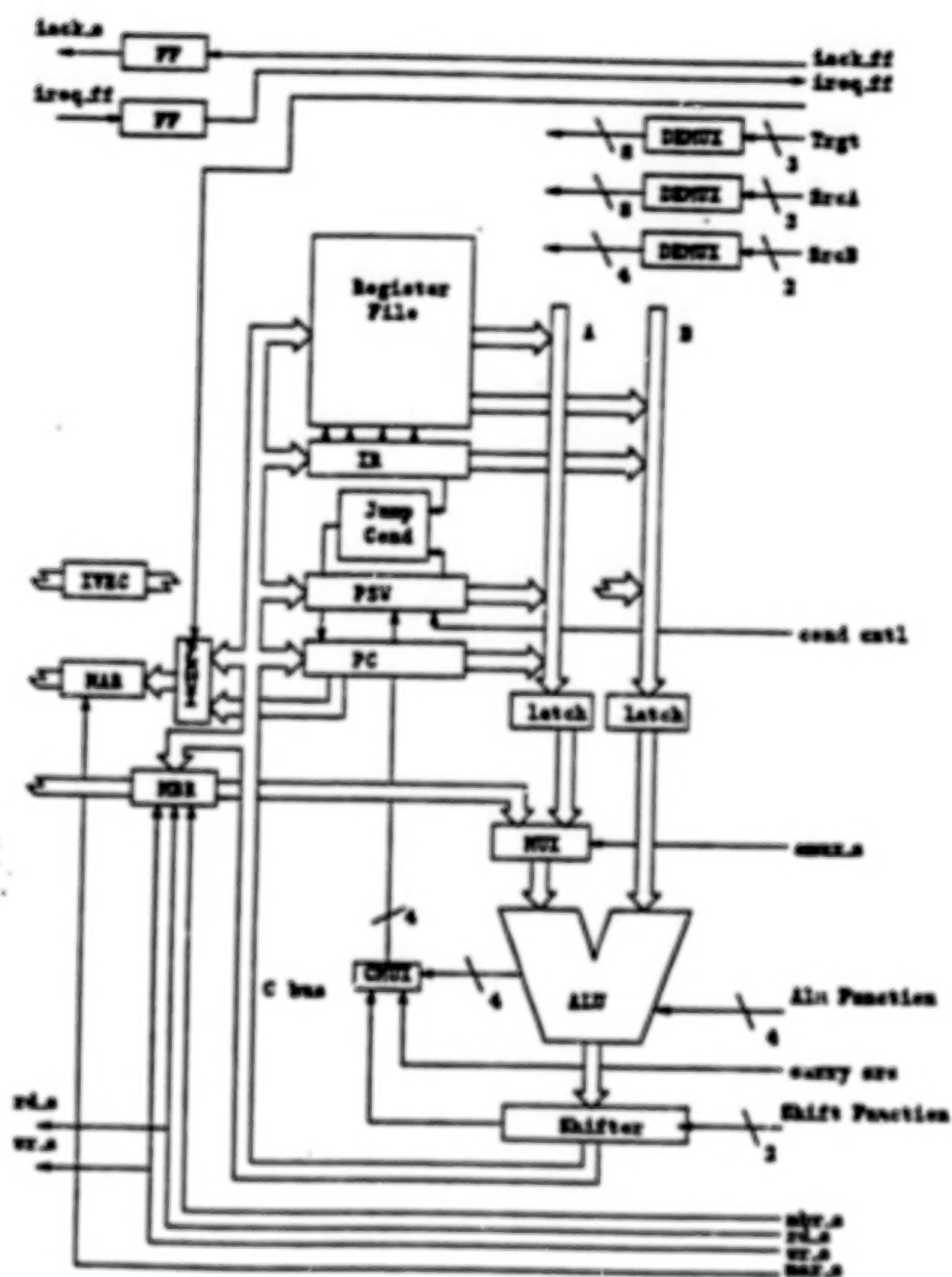


Figure 5.2: The AVM-1 Datapath

## *The Phase-Level Specification*

The  $n$ -tuple representing the state:

$$\mathbf{S}_{\text{phase}} = (\text{mir}, \text{mpc}, \text{reg}, \\ \text{alatch}, \text{blatch}, \text{mar}, \text{mbr}, \\ \text{clk}, \text{mem}, \text{urom}, \text{ireq}, \text{iack})$$

## *The Phase-Level Specification*

A typical function specifying an instruction's behavior from  $J_{phase}$ :

```
⊢def phase_two rep (mir, mpc, reg, alatch, blatch,  
                    mbr, mar, clk, mem, urom,  
                    ireq, iack) =  
  (mir, mpc, reg,  
   EL (bt5_val (SrcA mir)) reg,  
   EL (bt5_val (SrcB mir)) reg,  
   mbr, mar, (T,F), mem, urom, ireq, Iack mir)
```



## *The Electronic Block Model*

The electronic block model is not specified as an interpreter.

- EBM is a *structural* specification.
- The specification
  - is in terms of smaller blocks.
  - uses existential quantification to hide internal lines.

## *The Electronic Block Model*

```

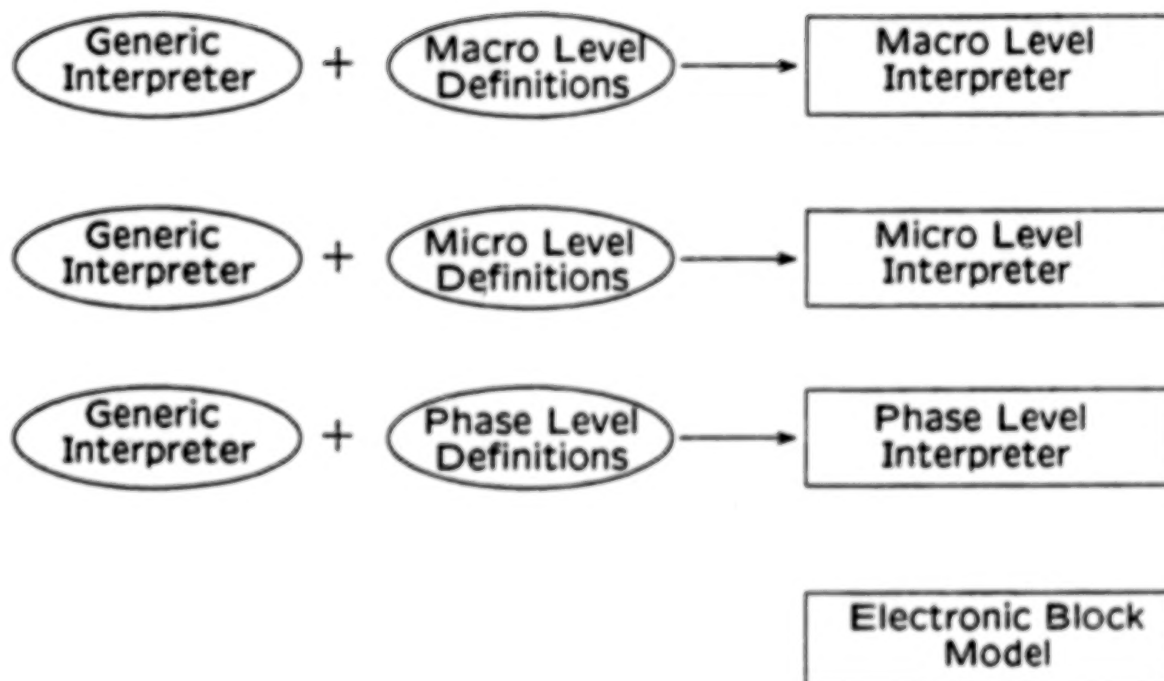
└ EBM rep (λ t. (mir t, mpc t, reg t, alatch t, blatch t,
                 mbr t, mar t, clk t, mem t, urom,
                 ireq t, iack t)) =
  ∃ opc ie_s sm_s iack_s
    amux_s alu_s sh_s mbr_s mar_s rd_s wr_s
    cselect bselect aselect
    neg_f zero_f (float:time->bool).
  DATAPATH rep amux_s alu_s sh_s mbr_s mar_s rd_s wr_s
    cselect bselect aselect neg_f zero_f float
    float ireq iack_s iack opc ie_s sm_s
    clk mem reg alatch blatch mar_reg
    mbr_reg reset_e ireq_e ∧
  CONTROL_UNIT rep mpc mir clk amux_s alu_s sh_s mbr_s
    mar_s rd_s wr_s cselect bselect aselect neg_f
    zero_f ireq iack_s opc ie_s sm_s urom
    reset_e ireq_e

```

Fully expanded, the electronic block model specification fills about six pages.

## *Generic Interpreters*

### Instantiation



## *Objects*

There are several abstract classes of objects that we will use to define and verify an abstract interpreter.

: *\*state*    An object representing system state.

: *\*key*      The identifying tokens for instructions.

: *time*      A stream of natural numbers.

We will prime class names to indicate that the objects are from the implementing level.

## Operations

<i>Operation</i>	<i>Type</i>
inst_list	$: (*key \times (*state \rightarrow *state))list$
key	$: *key \rightarrow num$
select	$: *state \rightarrow *key$
cycles	$: *key \rightarrow num$
substate	$: *state' \rightarrow *state$
Impl	$: (time \rightarrow *state') \rightarrow bool$
clock	$: *state' \rightarrow *key'$
begin	$: *key'$

## *Interpreter Theory* (obligations)

The *instruction correctness lemma* is important in the generic interpreter verification.

Here is the generic version of that lemma for a *single* instruction:

$$\begin{aligned} \vdash_{\text{def}} \text{INST\_CORRECT } s' \text{ inst} = & \\ & (\text{Impl } s') \Rightarrow \\ & \quad \forall t' : \text{time}'. \\ & \quad \text{let } s = (\lambda t. \text{substate}(s' t')) \text{ in} \\ & \quad \text{let } c = (\text{cycles}(\text{select}(s t'))) \text{ in} \\ & \quad (\text{select}(s t') = (\text{FST inst})) \wedge \\ & \quad (\text{clock}(s' t') = \text{begin}) \Rightarrow \\ & \quad ((\text{SND inst}) (s t') = (s(t' + c))) \wedge \\ & \quad (\text{clock}(s'(t' + c)) = \text{begin}) \end{aligned}$$

## *Interpreter Theory*

(obligations)

Using the predicate INST\_CORRECT, we can define the theory obligations:

1. The *instruction correctness lemma*:

EVERY (INST\_CORRECT  $s'$ ) inst\_list

2. Every key selects an instruction:

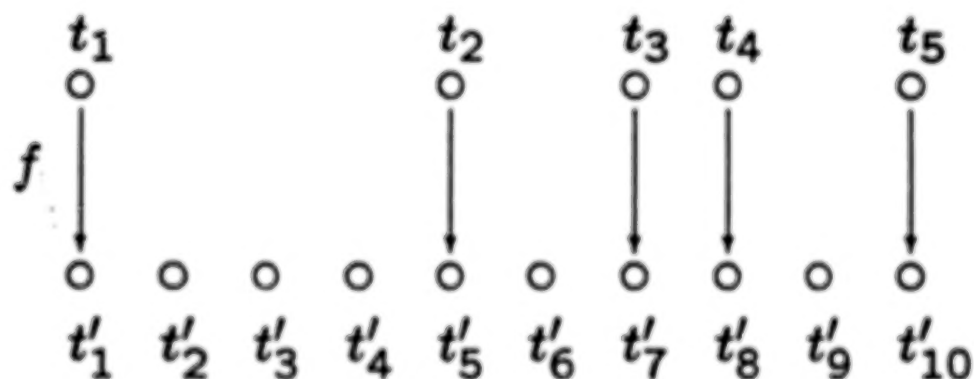
$\forall k : *key. (\text{key } k) < (\text{LENGTH inst\_list})$

3. The instruction list is ordered correctly:

$\forall k : *key. k = (\text{FST } (\text{EL } (\text{key } k) \text{ inst\_list}))$

## *Interpreter Theory* (temporal abstraction)

We need to show a relationship between the state stream at the implementation level and the state stream at the top level.



The function  $f$  is a temporal abstraction function for streams.



## *Interpreter Theory*

(definition)

An interpreter's behavior is specified as a predicate over a state stream.

$$\begin{aligned} \vdash_{def} \text{INTERP } s = \\ \forall t : \text{time}. \\ \text{let } n = (\text{key}(\text{select}(s \ t))) \text{ in} \\ s(t + 1) = (\text{SND } (\text{EL } n \ \text{inst\_list}))(s \ t) \end{aligned}$$

## *Interpreter Theory*

(correctness result)

Our goal is to verify an interpreter, **I** with respect to its implementation **M** by showing

$$\mathbf{M} \Rightarrow \mathbf{I}.$$

Here is the abstract result:

$$\vdash \text{Impl } s' \wedge (\text{clock}(s' \ 0) = \text{begin}) \Rightarrow \\ \text{INTERP } (s \circ f)$$

where

$$s = (\lambda t : \text{time}. \text{substate}(s' \ t)) \quad \text{and} \\ f = (\text{time\_abs } (\text{cycles} \circ \text{select})s)$$

## *Instantiating a Theory*

Instantiating the abstract interpreter theory requires:

- Defining the abstract constants.
- Proving the theory obligations.
- Running a tool in the formal theorem prover.

## *Future Work*

- New architectural features.
- Composing verified blocks.
- Verifying operating systems.
- Gate-level verification.
- Byte-code interpreter verification.
- Other classes of computer systems.

Building the mathematical part (window changes)

**StatusShell : trait**

**imports** AgcsStructure

**introduces**

**#.component :**

Window\_array  $\rightarrow$  Window\_status

md:  $\rightarrow$  Mode

knob : Value  $\rightarrow$  Action

**asserts for all** [agcs:AGCS\_state, ...]

**abbreviation**

agcs' == transition(agcs,act,sensor,plan)

agcs'.window.component =

**if** md  $\in$  agcs'.modes - agcs.modes

**then** chosen

**elsif** md  $\in$  agcs.mode - agcs'.modes

**then** current

**elsif** act = knob(i) **then** chosen

**else** agcs.window.component

Example: StatusShell{alt,alt\_eng,Airspeed}

## Design of the code:

- Packages `panel_logic`, `display_manager`, `sensor_data`, `flight_plan`, `flight_control`.
- State of `panel_logic` based on `AGCS_state`, etc.
- Actions  $\mapsto$  procedures of `panel_logic`:
  - read state of `panel_logic`, `sensor_data`, `flight_plan`
  - modify states of `panel_logic`, `display_manager`, `flight_control`
- Consistent with polling, interrupts, etc.

Specifying the code:

```
--| WITH TRAIT AgcsLogic, AgcsProperties,  
--|           LogicalDisplay  
--| WITH sensor_data, flight_plan,  
--|           display_manager, flight_control  
  
with sensor_data_types; use sensor_data_types;  
package panel_logic  
  --| BASED ON AGCS_state  
  --| INVARIANT  
  --| panel_logic.on -> good(panel_logic)  
  --| INITIALLY not panel_logic.on  
  ...  
  
end panel_logic;
```

```

procedure att_cws_switch;
--| WHERE
--|   GLOBALS IN   panel_logic
--|   GLOBALS OUT display_manager,
--|               flight_control,
--|               panel_logic

--|   IN panel_logic.on

--|   OUT panel_logic =
--|     transition(IN panel_logic,
--|               att_cws_switch,*,*)
--|   OUT FORALL ss: Sensor_state::
--|     look(display_manager,ss) =
--|     display(panel_logic,ss)
--|   OUT FORALL md:mode ::
--|     fc_engaged(md,flight_control) =
--|     engaged(md,panel_logic)
--|   END WHERE;

```



```
procedure turn_on_agcs
```

```
--| WHERE
```

```
...
```

```
--|   OUT panel_logic = initial_on_state
```

```
...
```

```
--| END WHERE;
```

477

518-61

319699

216

N91-17577

# Verification of Floating-Point Software

D. N. Hoover

Odyssey Research Associates, Ithaca NY

## Abstract

Floating point computation presents a number of problems for formal verification. Should one treat the actual details of floating point operations, or accept them as imprecisely defined? or should one ignore round-off error altogether, and behave as if floating point operations are perfectly accurate? There is the further problem that a numerical algorithm usually only approximately computes some mathematical function, and we often do not know just how good the approximation is, even in the absence of round-off error.

ORA has developed a theory of asymptotic correctness which allows one to verify floating point software with a minimum entanglement in these problems. We describe this theory and its implementation in the Ariel C verification system, also developed at ORA. We illustrate the theory using a simple program which finds a zero of a given function by bisection.

# **Verification of Floating-Point Software**

77051-107

**Douglas Hoover**

**Odyssey Research Associates, Inc.**

## **Difficulties**

- Machine real arithmetic does not have nice mathematical properties
- Doesn't match ideal arithmetic (overflow, round-off, underflow)
- Programs don't satisfy the specification we'd like them to

## **Asymptotic Correctness**

- Specify "ideal behavior" of the program (e.g. "program computes the square root of its input")
- Verify that if program is run on a sequence of machines converging to perfect accuracy, then program's behavior converges to ideal behavior

## Advantages of the Asymptotic Approach

- Machine real arithmetic can be specified loosely
- Specifications can be written in terms of ideal behavior
- Verification does not require roundoff error analysis
- Verifies *logical* correctness — absence of “bugs” from inaccuracy of machine arithmetic that are not related to error *magnitude*.

## Nonstandard analysis

$$\mathbb{R} \subseteq {}^*\mathbb{R}$$

Standard part map

$$st : {}^*\mathbb{R} \rightarrow \mathbb{R}$$

rounds off a finite nonstandard real to an infinitely close standard real.

### Continuity

$f$  is continuous at  $(a_1, \dots, a_n)$  if

$$st(f(a_1, \dots, a_n)) = f(st(a_1), \dots, st(a_n))$$

### Differentiation by algebraic manipulation

Let  $st(\epsilon) = 0$ ,  $\epsilon \neq 0$ . For all standard  $x$ ,

$$\begin{aligned} \frac{d(x^2)}{dx} &= st\left(\frac{(x + \epsilon)^2 - x^2}{\epsilon}\right) \\ &= st\left(\frac{2\epsilon x + \epsilon^2}{\epsilon}\right) \\ &= st(2x + \epsilon) \\ &= 2x \end{aligned}$$

## Nonstandard Analysis

- Asymptotic approach can be formalized naturally in nonstandard analysis using infinitesimals
- Primitive operations are assumed to return values which are infinitely close to the ideal values when the arguments and ideal answers are finite
- Programs are specified to have behaviors infinitely close to ideal behavior when inputs are finite



## Finding Roots of a Continuous Function

- `find_zero` searches for a root of a user-supplied function  $F$  by bisection.
- At each iteration, it tests to see if the values of  $F$  at the left endpoint and the midpoint are of opposite sign, and changes one of the endpoints to the midpoint so as to keep a root between the two endpoints.
- The program terminates when it finds a root or when it reaches a user-supplied bound on the number of iterations.

```

float find_zero(left0,right0,maxit)
float left0,right0;
int maxit;
{
    float left,right,center;
    float cval,lval0,rval0;
    int numit;

    numit = 0;

    lval0 = F(left0);
    rval0 = F(right0);

    left = left0;
    right = right0;
    center = (left + right)/2.0;
    cval = F(center);

    while(cval != 0.0 && numit < maxit) {
        if (lval0 * cval < 0)
            right = center;
        else
            left = center;
        center = (left + right)/2.0;
        cval = F(center);
        lval0 = F(left);
        numit = numit + 1;
    }

    return(center);
}

```

## Specification of find\_zero

IF  $F$  is continuous and find\_zero is started up with

- left0 and right0 not "large";
- maxit "large";
- $F(\text{left0})$  and  $F(\text{right0})$  of opposite sign

THEN find\_zero terminates normally (i.e. without an exception) and the value output is "close to" some zero of  $F$ .

## **Attempted Verification**

- Proof of termination is easy.
- Proof that termination is normal is a bit harder. Must prove that no overflow happens. To prove this, must prove that the values of the end-points stay in some range of numbers which are not "large".

How would we prove that the program returns an approximation to a root?

- Prove when the program terminates, the endpoints are "close". This follows from the fact that the program halves the interval a "large" number of times.
- Prove there's always a root between the endpoints. This should follow from the way the program decides whether to move the left endpoint or the right. From this we'd get center "close to" a root.

Unfortunately, it's not true that there's always a root between the endpoints.

## The Bug

- In the test statement, can have lval0 and cval of opposite sign, but have the product underflow to 0. This causes the program to move the wrong endpoint.
- Tests bear out this bug.

## Possible Fixes

Several ways to fix this bug

- Change test to

```
(lval0 < 0 && cval >= 0) ||  
(lval0 >= 0 && cval < 0)
```

- Change test so instead of always testing left endpoint against midpoint, it always tests the endpoint with the larger value of F against the midpoint.

This doesn't necessarily keep a root between the endpoints, but it delivers an approximation to a root anyway.

## **Ariel**

- Verification system for subset of C including real arithmetic and some UNIX system calls.
- Implements nonstandard formalization of the asymptotic approach.



## Semantic Verification

- Ariel verifies programs by generating a description of the program's denotation in a higher-order language (the *Clio metalanguage*)
- Specifications are statements about the denotation in the Clio metalanguage
- Verification is a proof of the specification directly from the description of the denotation in Clio theorem prover
- Specifications can be any statement about the program's denotation which can be expressed in the Clio, including termination

ORIGINAL PAGE IS  
OF POOR QUALITY

## C Semantics

- A "run" of the program is modeled as a sequence of *events*
- Events are:
  - the event of going into a certain state
  - terminating and returning a value
  - terminating and returning no value
  - raising an exception
  - an "unknown" event
- The semantics of the program is expressed as a collection of axioms saying which sequences of events can happen in the course of executing the program.

## **Sample Verifications**

- **ZBRENT** — a program which finds zeros of a continuous function by bisection
- **SWAP** — a very simple program to swap the contents of 2 locations which contains a surprising bug
- **HOSTILE BOOSTER** — a suite of programs, developed by Applied Technology Associates for SDIO, that estimate hostile booster trajectories. This verification is currently in progress.
- **SECURE DEVICE DRIVER** — specification and verification of security for an Ethernet device driver. Currently in progress.

495  
519-61  
319 700  
P-14  
V91-17578

## C Formal Verification with Unix Communication and Concurrency

D. N. Hoover

Odyssey Research Associates, Ithaca NY

### Abstract

This talk reports the results of a NASA SBIR project in which we developed CSP-Ariel, a verification system for C programs which use Unix system calls for concurrent programming, interprocess communication, and file input and output. This project builds on ORA's Ariel C verification system by using the system of Hoare's book *Communicating Sequential Processes* to model concurrency and communication. The system runs in ORA's Clio theorem proving environment. We outline how we use CSP to model Unix concurrency, and sketch the CSP semantics of a simple concurrent program. We discuss plans for further development of CSP-Ariel.

85251-107

## C Formal Verification with Unix communication and concurrency (NASA SBIR)

Aim: Verification system for

- C programs
- Unix system calls
- concurrent programming (fork, wait, exit, pipe)
- file and device i/o (read, write, open, close).

## Example program.

```
void producer();
void consumer();
int pipedes[2];

void main()
{
    int id;

    if (pipe(pipedes) == -1) return;

    id = fork();
    if (id == -1) return;
    if (id == 0) consumer();
    else producer();

    return;
}

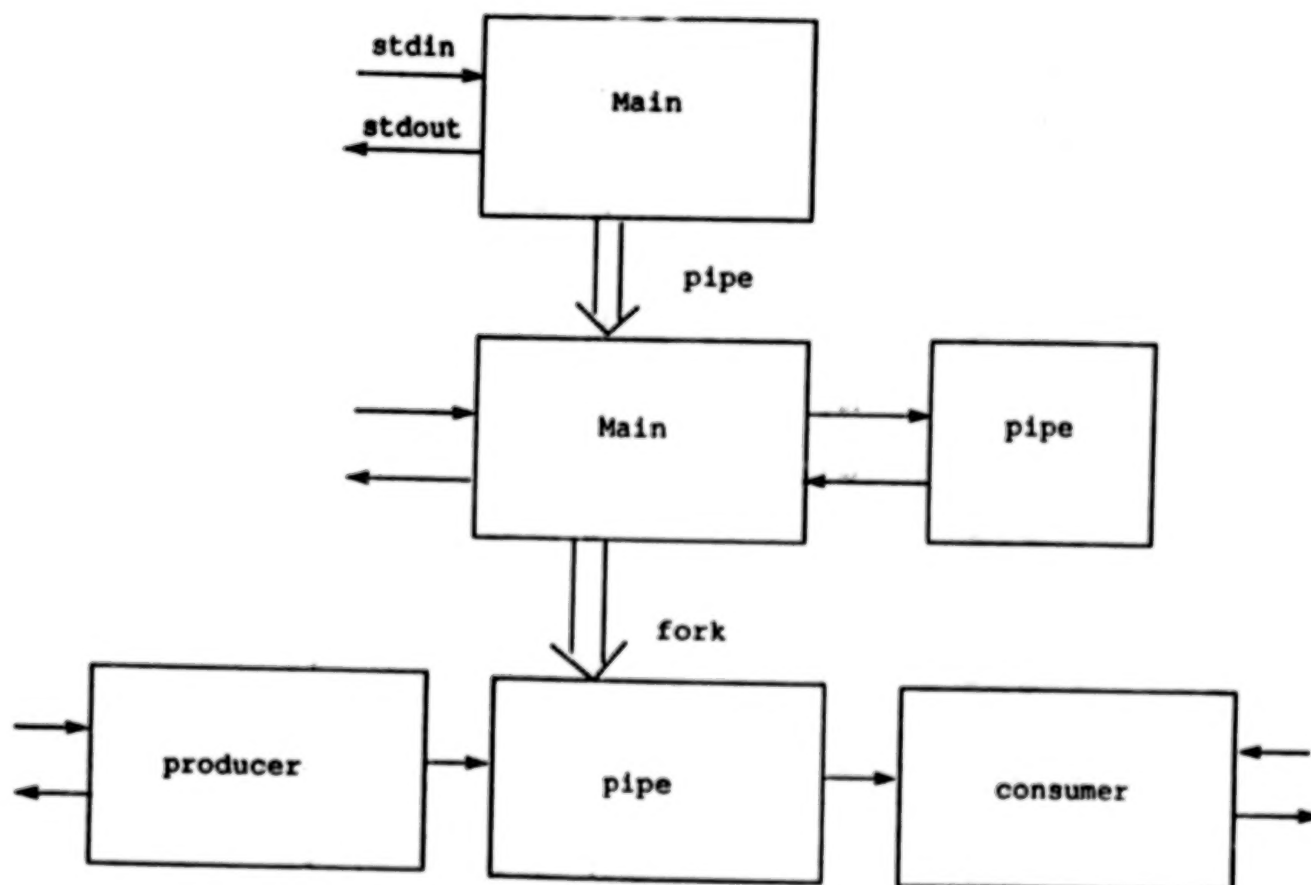
void producer()
{
    char c;
    int status;

    while (read(0, &c, 1) != 0) /* 0 = standard input filedes */
        write(pipedes[1], &c, 1);
    close(pipedes[1]);
    exit(wait(&status));
}

void consumer()
{
    char c;

    close(pipedes[1]); /* so that pipe read will fail when producer
                        closes its write end of pipe */
    while ( read(pipedes[0], &c, 1) != 0)
        write(1, &c, 1); /* 1 = standard output filedes */
    exit(0);
}
```

## Example Program Schematic



## Technical Approach

- C semantics via Ariel operational semantics (pre-existing)
- Unix communication and concurrency semantics via Hoare's CSP



## CSP (Communicating Sequential Processes)

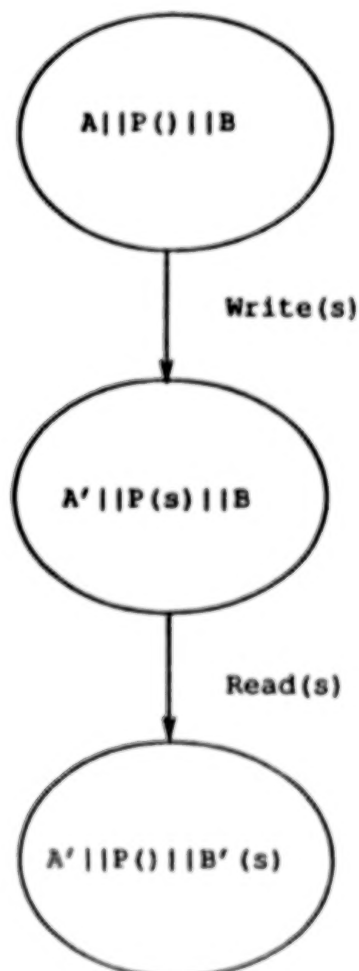
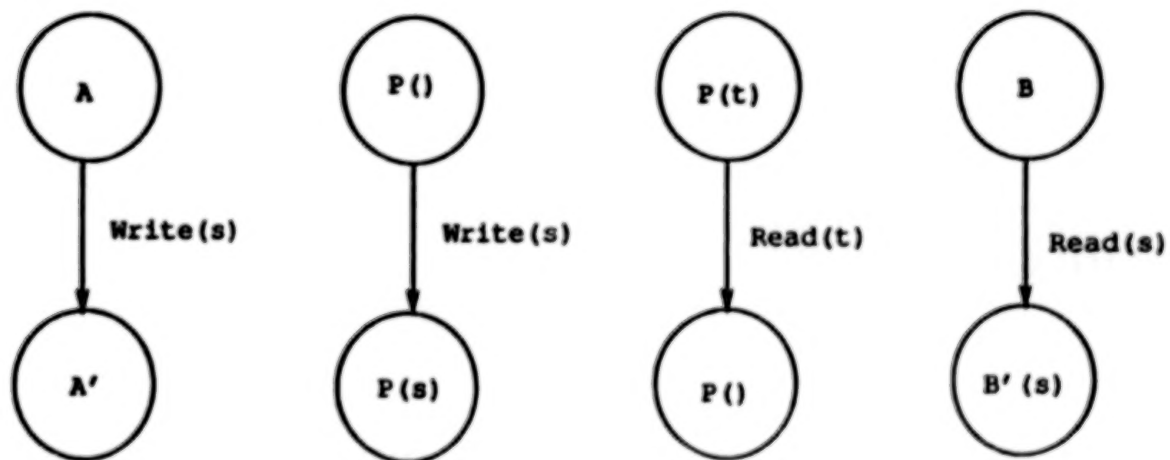
- See Hoare's book, *Communicating Sequential Processes*.
- An algebraic language for describing systems of processes with synchronous communication.
- Objects of the language are *processes* and *events*.
- Processes resemble state machines, events the input alphabet. Deterministic and nondeterministic processes.
- Processes participate in events and are transformed by them.
- Synchronous communication by participation in shared events.

## Unix modeling

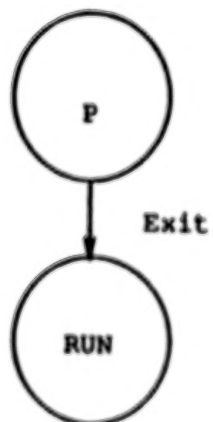
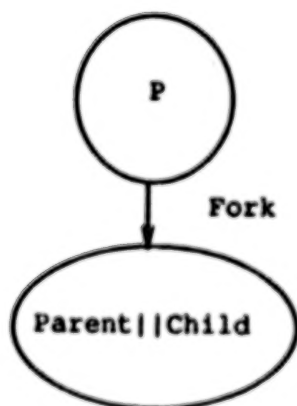
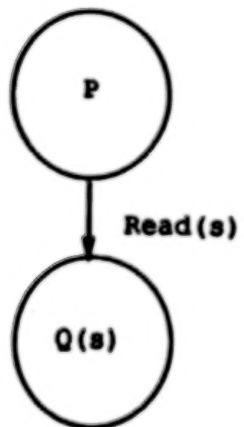
- Unix processes, files, pipes, and certain system tables are modeled as deterministic CSP-processes.
- Forking, pipe creation, file opening and closing, I/O, waiting, and exiting are modeled as events.

## Example: Asynchronous pipe communication

Sending process A, pipe P, receiving process B.



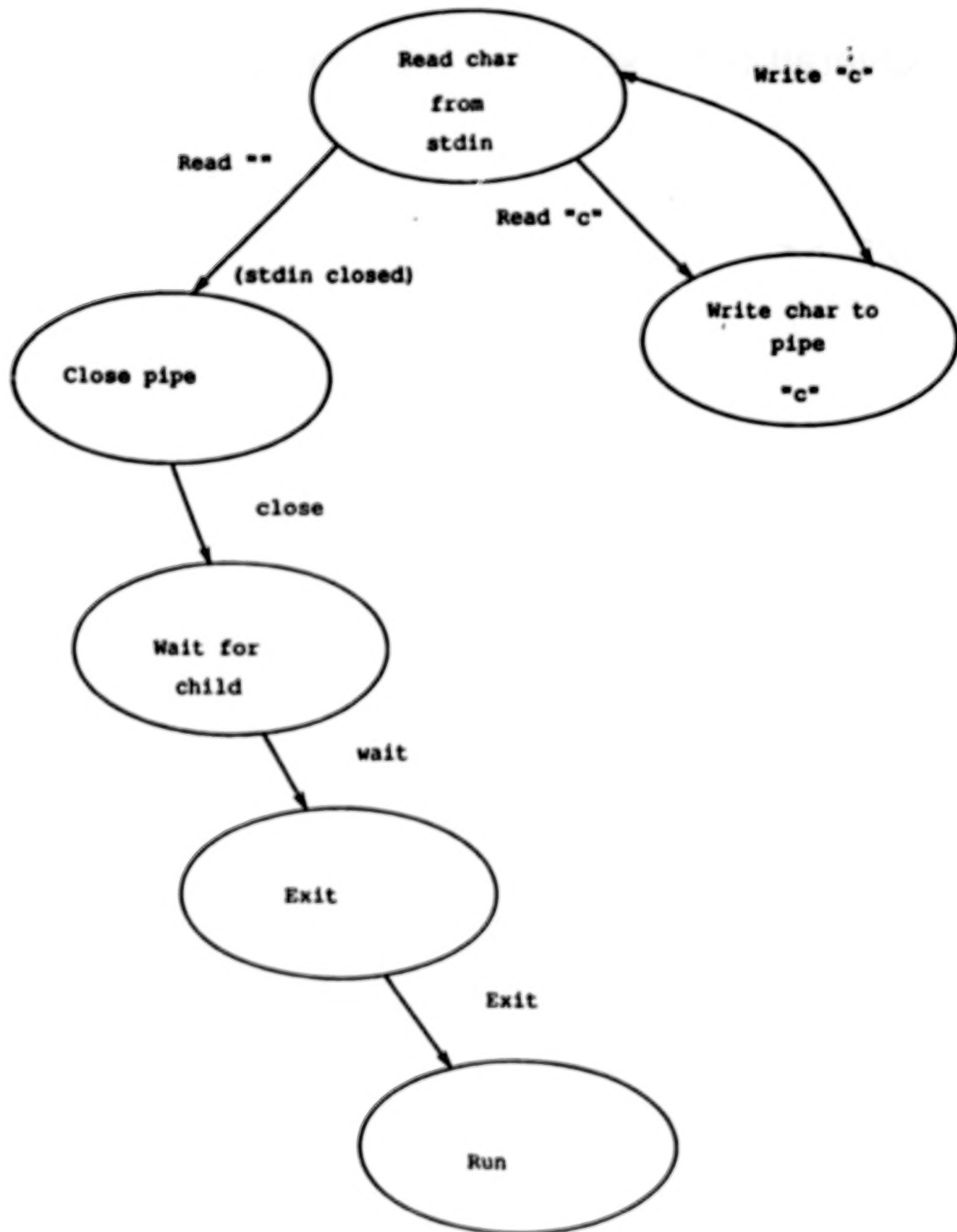
## Processes transformed by events



## Verification method

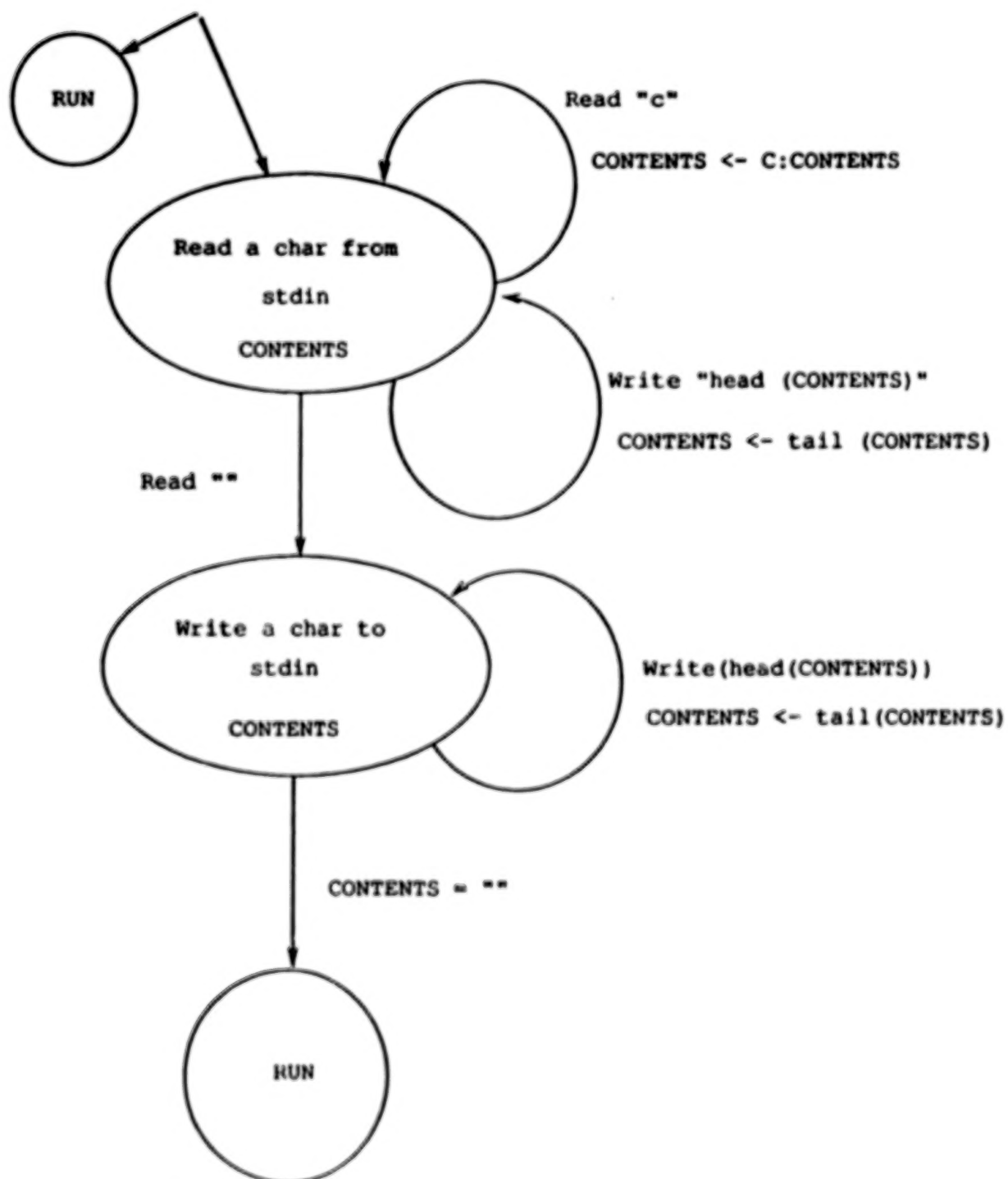
- C program given
- Ariel front end generates Caliban expression for abstract syntax tree of program.
- Ariel C semantics plus Unix system call semantics define denotation of a C program and associated files inside operating system as a CSP process.
- Internal operations of systems of processes hidden by CSP concealment operation.
- We reason about the resulting CSP process in Clio. Main tools are induction on traces (event sequences) of processes, and algebraic laws of CSP. Clio is a very general theorem prover, and we are not limited in the kinds of properties we can prove about processes.

## Producer as a CSP process



## Hiding events:

Overall process with non-I/O events hidden.



## CSP-Ariel Development Plan

- C semantics via Ariel symbolic interpreter (existing)
- Unix communication and concurrency semantics via deterministic CSP (initial work completed).
- Extensions to support network communication planned (sockets).
- Nondeterministic CSP and event concealment for specification and modularity (planned)
- Graphic specification support using Romulus interface (planned)



### Clio, Caliban, and, Ariel

- Ariel is a semantic verification system for a subset of C, written in Caliban and the Clio metalanguage. Floating point, overflow support via asymptotic correctness.
- Caliban is a lazy, purely functional language based on recursive equations and pattern matching.
- Clio is a higher-order logic theorem prover. Caliban is its term definition language. Clio's main proof methods are induction on Caliban definitions, term rewriting, and case splitting.



## Report Documentation Page

1. Report No. NASA CP-10052	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle NASA Formal Methods Workshop - 1990	5. Report Date November 1990	6. Performing Organization Code
7. Author(s) Ricky W. Butler, Compiler	8. Performing Organization Report No.	10. Work Unit No. 505-66-21-01
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225	11. Contract or Grant No.	13. Type of Report and Period Covered Conference Publication
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001	14. Sponsoring Agency Code	
15. Supplementary Notes This workshop was organized and chaired by Ricky W. Butler of NASA Langley Research Center.		
16. Abstract This report documents a workshop in Formal Methods held at the NASA Langley Research Center on August 20-24, 1990. The workshop brought together researchers involved in the NASA formal methods research effort for detailed technical interchange and provided a mechanism for interaction with representatives from the FAA and the aerospace industry. The workshop also included speakers from industry to debrief the formal methods researchers on current state of practice in flight critical system design, verification, and certification.  The goals of the workshop were: (1) Define and characterize the verification problem for ultra-reliable life-critical flight control systems and the current state of the practice in industry today, (2) Determine the proper role of formal methods in addressing these problems, and (3) Assess the state of the art and recent progress toward applying formal methods to this area.  Attendees included NASA personnel, researchers from the four supporting contract organizations, RSRE personnel, invited speakers, and representatives from other government research organizations with interests in formal methods.		
17. Key Words (Suggested by Author(s)) Formal Methods Digital Flight Control Verification Design Proof	18. Distribution Statement Unclassified - Unlimited  Subject Category 61  Flight Control Control Systems Design Logic design Fault Tolerance Digital Systems Conferences  Architecture (computer) logic circuits computer systems design	
19. Security Class. (of this report) Unclassified	20. Security Class. (of this page) Unclassified	21. No. of pages 512
		22. Price A22

ORIGINAL PAGE IS  
OF POOR QUALITYORIGINAL PAGE IS  
OF POOR QUALITY